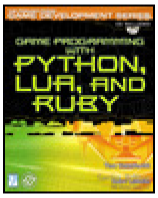


Game Programming with Python, Lua, and Ruby
By Tom Gutschmidt

Publisher: Premier Press
Pub Date: 2003
ISBN: 1-59200-079-7
Pages: 472

LRN

Dedication

This book is dedicated to Hailey and Sidney, the two biggest game players in our household

Acknowledgments

I would like to thank, in no specific order, the following individuals:André LaMothe and the staff at Premier Press—especially my editors Emi Smith, Mitzi Koontz, and Estelle Manticas. I would also like to thank my past editors Todd Johnson and Kieron Murphy.

I want to thank my parents, Katherine and James, for being so supportive over the years. Thanks also to my sister Tanya and her husband David, as well as to the rest of my immediate family—Alex, Raleigh, Steve, Stephanie, May Lou, Dodie, Dick, Bobbie, and Sophie—for their encouragement.

I want to especially thank my loving and wonderful wife Susan for putting up with my writing mood swings and geek chattering, and for being so kind when I was slumped over a desk for months in a dark office, furiously typing and staring into the cold blue monitor. Next year I'll try getting some sun.

About the Author

Thomas Gutschmidt has been professionally involved in the computer industry for the past seven years and currently works for a large software company headquartered in Redmond, Washington. He has been a freelance author and writer for three years and has been involved in several open source game projects and module development projects. He currently lives in the Northwest with his wonderful wife, Susan, their four cats, two rats, several goldfish, and the occasional urban wildlife refugee.

Letter from the Series Editor

Game development has reached a fever pitch in the past couple of years— photorealistic rendering, advanced physics modeling, a million-plus polygon worlds—and multiprocessor consoles and PCs are powering the revolution. At the same time, experimentation with scripting languages to help control the high-level aspects of games has gone from a convenience to an absolute necessity. No longer can game programmers think of something as absurd as writing a game in C/C++. Game engines may be written in C/C++, but games—no longer. Today's state-of-the-art games are controlled almost exclusively with scripting languages.

In the past, scripting languages were custom-made or derivative works made up of the C/C++ compiler and creative use of the pre-processor. Times have changed, and today developers are faced with a number of potential scripting languages to use in their games. Notable players are Python, Lua, and Ruby. Each of these languages has advantages and disadvantages, but any of them can do the job. Game Programming with Python, Lua, and Ruby takes you on a tour and tutorial of each language, highlighting its strengths and weaknesses and offering you detailed examples of getting each language up and running and interfaced to your game and host languages, such as C/C++.

With Game Programming with Python, Lua, and Ruby, you won't spend a lot of time learning irrelevant material—instead, you'll get just the information you need. Tom Gutschmidt delivers a non-biased view of each language and gets you up and running as soon as possible in each of the most popular scripting languages today—Python, Lua, and Ruby.

Sincerely,



André LaMothe

Series Editor

Introduction

This book is unusual because it covers game programming in three different scripting languages in three separate sections. Python, Lua, and Ruby are wonderful languages used all over the world to build efficient, flexible, scalable, and well-integrated programs and systems.

For the same reasons that these languages have been great choices for other projects, Python, Lua, and Ruby also are great for making games. This discovery, in fact, was made over a decade ago. Chances are you've played a computer game that utilized one of these languages during development. You may be currently working through game levels that were designed with Lua, or playing on a graphics engine prototyped in Python, or using an Internet ladder developed with Ruby.

Introduction

This book is unusual because it covers game programming in three different scripting languages in three separate sections. Python, Lua, and Ruby are wonderful languages used all over the world to build efficient, flexible, scalable, and well-integrated programs and systems.

For the same reasons that these languages have been great choices for other projects, Python, Lua, and Ruby also are great for making games. This discovery, in fact, was made over a decade ago. Chances are you've played a computer game that utilized one of these languages during development. You may be currently working through game levels that were designed with Lua, or playing on a graphics engine prototyped in Python, or using an Internet ladder developed with Ruby.

Why Learn Another Language?

Programming languages require a lot of discipline to learn. They each have their own set of formal specifications. They all have unique ways of handling data, data structures, reference mechanisms, and command flow. And underneath all this they each have their own design philosophy. So the question arises: "Why would anyone want to learn a new programming language, let alone three new programming languages?"

Well, first of all, these three high-level languages are great starting places to learn programming. For the most part, they are cleanly designed, well documented, and very kind to new programmers. Despite this, they are not toys. They are flexible and powerful, suited for both large projects and classroom exercises.

Second, every language has its own strength and weaknesses. The decisions you must make during software and game development become easier when more than one possible tool is available to you. In an ideal development environment, problems are solved in a general way and then the best language for a particular job is chosen. It may be difficult in tomorrow's job market for a programmer to get away with knowing only one or two languages well.

Finally, these three languages are really very similar. Much of what you learn from one will be applicable to the others. The more languages you learn, the easier the next one will be to pick up. This compound learning curve eventually begins to work greatly to your advantage, and after enough experience you will get to the point where you can learn a new language in days, simply by relating what is in a given manual to what you already know.

What's on the CD-ROM?

The CD that comes with this book is designed to launch automatically when inserted into a machine running the Windows operating system. On the CD is the source code for all of the samples and programs written in the book. These are separated into folders organized by chapter.

The CD also contains the software necessary to install Python, Ruby, or Lua on your system. This software is also separated into different folders—a Python folder, a Ruby folder, and a Lua folder.

Also on the CD are several open-source libraries and utilities that are either used for the source code samples or as examples in this book. These includes PythonWin, Distutils, Numeric Python, PAWS, Py2Exe, Pygame, PyOpenGL, Pyzzle, RubySDL, LuaSDL, and Clanruby.

Part ONE: Introducing High-Level Languages

Welcome to the first part of this book! In this part, I'll be introducing high-level languages and covering some of their parallel features. I'll introduce Python, Lua, and Ruby, but I'll save the gory details for the later parts of the book. Part One is a gentle introduction to these languages' features, syntax, and similarities, as well as to their cohorts and partners in the gaming industry. 7

Chapter 1. High-Level Language Overview

All programmers are playwrights, and all computers are lousy actors.

——Unknown, quoted by Michael Moncur in The Quotations Page

Where to start? There is much to cover, and we have a very short time together. This is Chapter 1 of Game Programming with Python, Lua, and Ruby. In this chapter I'll discuss the specific pros and cons of programming with these high-level scripting languages (after explaining what a high-level scripting language is, of course), delve into their properties and history, and then wrap up the chapter with a listing of some of the major projects these languages are responsible for.

High-Level Language Roots

In the beginning, a programmer needed to know everything about the internal workings of a specific computer in order to program it. This took quite a bit of knowledge and effort. Then, from within the programming industry, an idea emerged. The idea was to reduce the amount of knowledge of the internal workings of the computer a programmer needed to write programs (some call this idea encapsulation). If adopted, this concept could make it easier and faster to program, and the program itself could be less error prone. A second idea followed this first one: If programs could be presented in a familiar language, then programmers could learn them quickly. These ideas eventually led to high-level languages.

High-level languages were created to make programming easier, but today's high-level programming languages have seriously evolved from early predecessors like FORTRAN in the 1950s. You have your high-level languages, your high-level scripting languages, your high-level open-source scripting languages, your high-level open source object-oriented scripting languages, and your very high-level open-source object-oriented scripting languages (yes, the dreaded VHLOSOOSLs). So much for easier. Despite the long, often buzzword-filled names, there are those of us who love these languages. And luckily we like to spend time explaining why.

Before you commit to a project with a certain language, spend some time under the hood, read a book or two, and check into the language's community. Most good languages will already have a large and very active user base—that is, if they have useful features that appeal to a wide audience and if they are capable of getting the job done. This chapter and the next spend a bit of time showing how Python, Lua, and Ruby appeal to a wide range of jobs and professionals and how their communities have grown in power and presence in recent years.

NOTE

Open Source Software

The basic definition of open source software is software that has its code base opened up and viewable to users. Anyone can look under the hood of open source software to see how it works.

Open source software likely originated with the United States government. In the 1960 and 70s, the U.S.was funding systems of distributed computers that would later become the Internet, and they actively encouraged scientists to develop technologies that could facilitate distributed computing. Academic researchers, including those at MIT, UCLA, Berkeley, and Stanford, and later corporate researchers at companies like IBM and Xerox began developing technologies for computers and operating systems to communicate with each other. Out of this movement came utilities such as Sendmail and TCP/IP. Other tools, like Emacs, Perl, and Linux, followed.

Open source does not necessarily mean "free." Open source code is usually free to download, view, and modify, but most open source software is copyrighted and possesses some sort of license. Often there are restrictions on its use. For instance, many open source licenses require that if modifications are made to source code, the

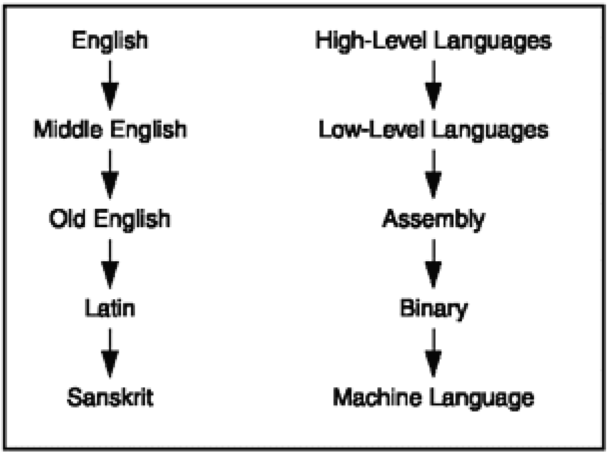
modifications need to be released to the public at large. This means that open source utilization in private, commercial software development involves other costs. Of course, using commercial software also involves software licenses and tracking copies and usage.

How Programming Languages Work

Let's ignore Webster and Oxford and pretend that the word language simply refers to a system used to communicate. Languages possess syntax, and syntax defines the order, arrangement, or structure of the system of communication.

This book is written in English, a language sometimes referred to as American or Present-Day English, which evolved from the Early Modern, Middle, and Old English languages. Some historians and linguists claim that forms of English can be traced through Gothic, Latin, and Greek, eventually finding roots in Sanskrit (see Figure 1.1).

Figure 1.1. A comparison between English and computer language roots



If you trace programming languages back to their source, you'll find that all computers, or at least their CPUs, have an internal machine language that they execute directly. Internally, all data in a modern digital computer is stored as binary on and off states. The tools used to manipulate these on/off states are coded in a binary representation and normally consist of operation codes and addresses. The operation code indicates which operation is to be carried out, while the address dictates the memory location. The operation basically amounts to what, and the address basically amounts to where. This process looks something like Table 1.1. Given the operation codes and address in Table 1.1, a programmer can enter in the instructions in pure binary form as:

```
00100010 10010101
```

Table 1.1. Sample Machine Language Instructions

Procedure	Binary	English Translation
Operation	00100010	means "load(X)"
Address Location	10010101	means "location 13" on the CPU

These instructions would load X into location 13. As you can imagine, it is very tedious to write this way. A programmer needs to be especially careful to keep track of which address locations he is using to store data; program errors often lead to operations overwriting the wrong addresses.

Programming languages express these operations and addresses at a higher level of logic than the low-level CPU code. They are translation systems that allow a computer and a person to communicate with each other in a medium that is something between English and CPU binary. With a programming language, a person can program what actions a computer will take and the types of data the program acts upon without having to speak the computer's language.

CPU is an abbreviation for central processing unit. Often referred to as the processor or central processor, the CPU performs most of a computer's calculations. CPUs are normally one or more printed circuit boards, but may be housed in a single chip called a microprocessor. CPUs typically consist of an Arithmetic Logic Unit (ALU) that performs logical operations and a Control Unit that extracts instructions from memory and decodes and executes them.

Low-Level Languages

Each CPU has its own unique machine language, which consists of binary numbers only. Machine languages are tedious and repetitive, two things that humans are poor at and seem to dislike universally. These machine languages are low-level languages. Low-level languages closely reflect the inner workings of a computer and are sometimes referred to as machine-oriented languages.

The most prominent example of a low-level language is assembly. Assembly language is one step higher than machine language and consists of numeric instructions for a specific computer architecture. Assembly is limited because it needs detailed instructions, and there isn't much portability from platform to platform.

In assembly, machine language commands are replaced by mnemonic commands on a one-to-one basis. An assembler program then takes care of converting the mnemonic into corresponding machine language binary. In assembly, a programmer can also use symbolic addresses for data items. The assembler program will assign these symbolic addresses to machine addresses and make sure they do not overlap or overwrite each other. Today, most assembly programming is reserved for high-end performance device drivers, where execution speed and code size are more important than rising development costs.

In the early days of games, assembly was the mainstay, and common game platforms were MS-DOS, Apple, and the Atari 800. But as game programs grew in size, programmers found that assembly was pretty poor at scaling, and as code grew programs became exponentially more difficult to maintain, and testing and debugging them became more and more difficult.

After assembly languages came compiled languages like C, COBOL, and FORTRAN. With a compiled language, the programmer writes source code, and then a compiler takes the source code and translates it into machine language for a particular computer. With a compiler hard at work, the programmer can ignore some of the machine-dependent details, and with a good compiler the program will run almost as fast as with assembly.

C in particular really made large-scale programming possible by automating much of what programmers found difficult in assembly. C also universalized the idea of functions, so for the first time programmers could share functions they wrote with each other. This led to larger development teams and a growing pool of development tools. Great games came out of C (and still do), like Doom and X-Wing.

Today's High-Level Languages

The terms high-level, interpreted, and scripting all share a similar conceptual space when it comes to programming, and this often causes confusion. Over the next few pages I'll explain each term. Pay attention—there may be a quiz coming up!

High-level languages are designed with the native language of the programmer in mind. They are sometimes referred to as problem-oriented languages and are often very specific in focus. BASIC is a good example of a high-level language; it was designed for first-time programmers as a learning tool. COBOL and FORTRAN are other good examples. COBOL was designed for business problems, and FORTRAN for solving scientific and mathematical problems.

NOTE

Python is sometimes referred to as a "Very High Level Language" (VHLL). This term appeared in the mid 1990s to describe languages used for rapid prototyping. Two features that supposedly separate VHLLs from your standard high-level language are dynamic types and an interactive environment that allows you to make changes without having to go through the entire relink recompile steps.

Instructions in high-level languages closely resemble everyday language, making high-level languages much easier to learn and use than their low-level equivalents. The programmer does not need to have detailed knowledge of the internal working of the computer in order to program instructions. Each instruction in high-level is equivalent to several machine code instructions that then are either compiled or interpreted to translate them into machine code.

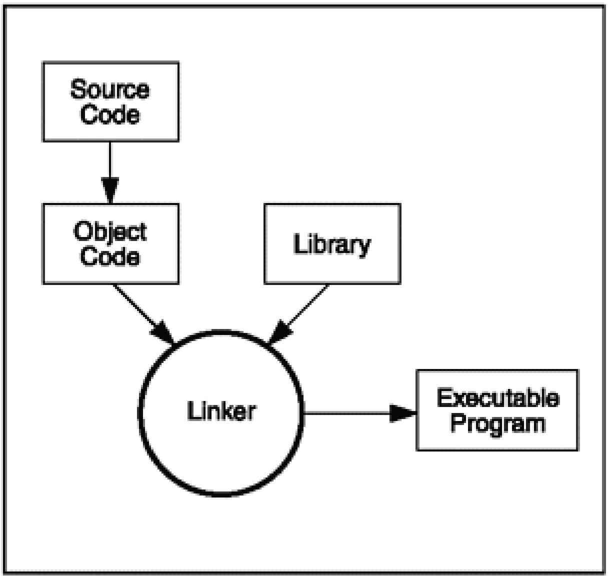
Interpreted versus Compiled Languages

A high-level interpreted language translates the programmer's written code step-by-step at runtime, or when the program is actually running. A high-level compiled language translates a programmer's written code before the program is run, a process normally called compiling. This changes the written code into an executable or object-code that can then be run as a program on a computer.

Many modern programming languages allow themselves to be both interpreted and compiled, but normally a particular language is more suited to one or the other. AWK, Perl, and Python are examples of interpreted programming languages. BASIC, COBOL, C, and FORTRAN are examples of compiled programming languages.

When a program is compiled, the compiler takes the source code files and generates object code with those files. The object code is then wrapped together during a linking process to produce an actual executable. This process is illustrated in Figure 1.2.

Figure 1.2. The process of compiling source code into an executable file or program



When comparing the two types of languages, you can usually make two generalizations. The first is that interpreted programs are usually much slower than their compiled counterparts (although the actual process of compiling may take quite a bit of time as well). The second is that interpreted languages are more flexible at runtime than compiled languages because they can interact with the execution environment. In other words, in order to gain flexibility, you must slow down.

Scripting Languages

Scripting is a term used to denote the scripting of a computer, akin to an actor who follows a script to perform a play. A scripting language is a high-level language used to assemble components into a predefined software architecture. Scripting languages, sometimes called glue-languages, are designed for scripting the operation of a computer. Normal operations that would be considered scripting are administrative tasks such as running automatic backups, text processing, running server-side requests such as CGI processing, or automating software tests. Python, Lua, and Ruby are considered scripting languages in one form or another, as are ASP, AWK, JavaScript, Perl, and VBScript.

The scripting-language family is hard to pin down. VHLL languages include the various types of UNIX shell command-line interpreters, and even languages like AWK, Perl, and Lisp can be classified as scripting languages. Unfortunately, there is no universally accepted definition of what a pure scripting language actually is, but they usually have most of the following features:

- They are interpreted languages.
- They possess a simple syntax.
- Variables are dynamic, so that they can act as strings or numbers, depending on what operation is being performed on them.

- Variables are created when referenced, as opposed to being allocated to memory early on or during compile time. Internal details about the variables are only resolved when necessary. This process is sometimes called late binding.
- They possess high-level string manipulation features. Concatenation and searching are built into the language.
- They do not possess pointers.
- The programmer does not handle memory allocation; the language handles it automatically.
- Garbage collection (release of unused memory) is handled automatically.
- The language is interactive and can give feedback while running, often pointing out errors, mistakes, and problems.
- The code is stored in a plain text format.
- Processor and operating system independence exists, and the code can work in many different environments.
- They simplify the usage of common commands such as array sizes, data types, or expressions. Common commands are often built in.

  NOTE

  Statically versus Dynamically Typed Languages

  The specific system by which data is organized in a program is called the type system. There is an actual discipline devoted to the design and study of type systems, called type theory. In practice, however, there are normally only two type systems: static and dynamic.

  Statically typed languages need predefined types for pieces of data, and values can only have one fixed typed. Static systems are sometimes called type-safe or strongly typed. C++ and Java are examples of statically typed systems.

  Dynamic systems treat data locations interchangeably. They are sometimes called latently typed systems. Again, the key here is flexibility versus speed. Dynamic systems are a bit slower during runtime than their static brethren, but they are faster to code, as there is no need to predefine variables or check for buffer overflow against them. Examples of dynamically typed systems include Lisp, JavaScript, TCL, and Prolog.

- Statements are usually terminated by returns or new lines, rather than with semicolons or punctuation.
- They are optimized for programmer efficiency as opposed to program efficiency.
- They are optimized for text manipulation, data filtering, system applications, and/or building graphical user interfaces.
- Components of foreign code, such as shell commands, other language libraries, or COM, can be embedded or "glued" to the scripts, and the language provides interfaces to external components. This process is called extensibility.
- They are considered a rapid prototyping language.

The Pros of High-Level Languages

The cost of software is determined by the time it takes to test, debug, modify, and maintain a code base. In a not-too distant past, the programming field was a much more static one. Programming was done in a controlled hardware environment, and things like testability, modification, and portability weren't as important.

High-level languages exist because human time is important. Often the loss of computer runtime efficiency will gladly be traded for actual savings in human labor. The code base for a project using a high-level language might be three times shorter than Java and five times shorter than C++.

Probably the biggest problem with low-level code is that adapting it to different architectures (platforms) can be problematic. If you cut C off from its standard compilers and libraries, it is pretty much incapable of porting to a different architecture. Generally, low-level code has to be rewritten for each specific platform.

High-level languages, on the other hand, are very portable, only needing an alteration to the interpreter or compiler for the new platform—or needing nothing at all. Compilation time is usually short—measured in seconds sometimes. Human-time debugging on a new platform, especially in a low-level language like assembly, can easily take weeks. This is an obvious trade off.

Another big benefit of high-level languages is reusability. High-level code can be crafted into small components that are easy to use, as well as easy to organize and bring into future projects. Such modularity promotes the creation of formal and informal code libraries. Most high-level languages have particularly great libraries for putting together graphical user interfaces.

Higher-level languages have more human readable words and phrases and fewer abstract symbols, peculiar syntax, and abbreviations. This can make them easier to write and maintain. This makes testing, debugging, and modifying an easier task. Most importantly, it makes reading them easier, a boon for the high turnover world of software development.

Safety in source code is a big issue these days. Many high-level language features have the interesting side effect of producing more secure, bug-free code.

Take, for instance, buffer overruns. A buffer is a device or structure that holds data. Buffer overruns occur when someone overflows a buffer by giving it more data than it can handle. A simple example is a login prompt to a computer or Website. The programmer who develops the login expects that most login names will not be more than eight characters long and gives the buffer that holds the login data enough space in memory to hold eight characters. But then some malicious user comes along and writes 257 characters to the login. If the buffer and input login haven't been specifically designed to handle such a case, the software will fail. Worse, it could allow the user the ability to write data somewhere besides the login prompt.

CERT (the Computer Emergency Response Team) reports that a majority of bugs and exploitable holes in software (majority meaning as high as 80 percent) are caused by

simple buffer overruns. This type of exploit is very common because manually coding pointers and garbage collectors can be a very buggy and error-prone enterprise. High-level languages normally take care of these tasks automatically for the programmer. Taking away the manual process of handling pointers, automatically handling garbage collection, and assigning memory allocation of variables at runtime makes it difficult to cause buffer overruns.

My favorite feature of high-level languages is that they are easy to learn—so easy, in fact, that they are often considered fun. High-level languages are particularly suited for applications in which:

- The main focus is to connect existing components.
- A GUI is required.
- A lot of string manipulation is required.
- You expect the application's functions to evolve rapidly or change quickly over time.

Cons of High-Level Languages

How high-level can a language get, and what are the potential problems associated with them? In Star Trek, science-fiction computers communicate with their commanders in an almost human language. Our science fiction tells us that the higher-level a language is, the easier it is to communicate, the better. In real life this isn't the case.

The biggest problem with high-level languages is that they are slower than their low-level counterparts. There is a give-and-take relationship between the speed of development and the efficiency of a program. C is speed efficient because the programmer handles all of the low-level resource management by hand.

Since they aren't as speedy and they handle low-level resource management themselves, high-level languages are not great for engineering system–level programs like device drivers or kernels, or other situations in which you need tight control over low-level tasks, like memory allocation. Lack of speed also makes them poorly suited to computationally intensive applications, like those that build data structures and algorithms from scratch. In particular, a low-level language may be more suited to your application if:

- It needs to implement complex algorithms or data structures.
- It needs to manipulate large data sets.
- Execution speed is critical.
- The functions are well defined and will not change.

The pros and cons of high-level languages are highlighted in Table 1.2.

Table 1.2. High-Level Language Pros and Cons

Pro	Con
Saves human time	Less efficient during computer runtime
Portable to many platforms	Specific platforms aren't as efficiently utilized
Modularity and reusability	Can lead to dizzyingly high number of libraries
Easier to read, write, and maintain	Loss of some control over code organization
Auto-management of many bug-prone features	Less low-level control of resources
Easy to learn	Too many programmers could lower one's salary!

NOTE

High-level languages are criticized more often for their lack of speed than anything else. But keep in mind that they usually can be compiled or semi-compiled. This can make

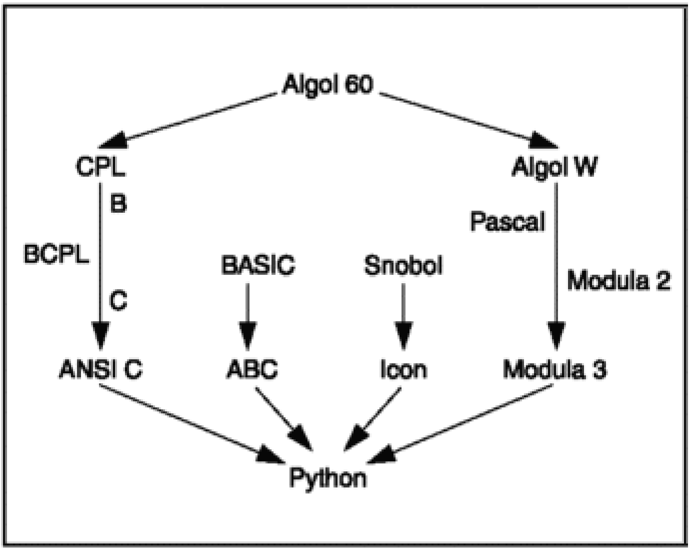
them much faster than languages like Perl, AWK, or other rivals. Also, today's machines are 500-2000 times faster than their predecessors from the 1980s.

A Brief History of Structured Programming

In the1960s software development went through a number of growing pains. Development schedules often ran far behind predictions, costs were much higher than projected, and often the end software product was unreliable or buggy. People began to realize that software development was extremely difficult, and some folks began to research development methods of this new field to see what could be improved. Out of this research came the concept of structured programming.

Structured programming is a method of programming designed to help make large programs easier to read and is a predecessor to Object-Oriented Programming. Structured programs are usually illustrated in simple graphs that have a top-down approach and flow. Figure 1.3 illustrates a structured-programming graph in which the circles represent starting and ending points, the squares represent program blocks, and the diamonds represent branches.

Figure 1.3. An illustration of a simple structured language



NOTE

Object-Oriented Programming

Object-Oriented Programming (or OOP) is actually a design methodology that defines programs in terms of objects. Objects are entities that combine both state (data) and behavior (methods). In pure OOP, programs are sets of objects that communicate with each other to do various tasks. This is a pretty different design than procedural languages (the standard before OOP), where data and procedures are separated.

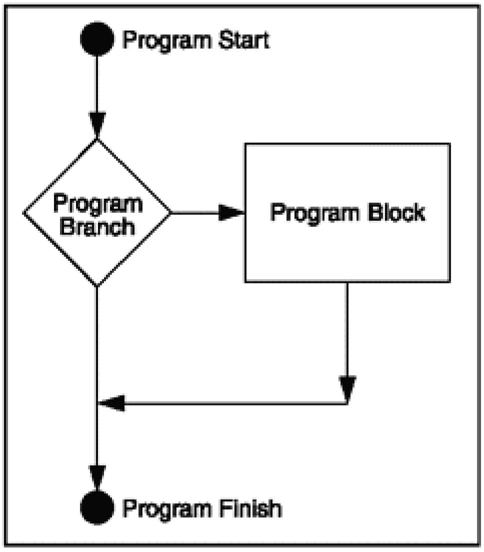
Unfortunately, there is some disagreement about exactly what features are required to qualify a programming language as "object-oriented," so giving a definitive description of an OOP language is difficult. Traditionally, the first OOP language is considered to be Simula 67, whose OOP features were later refined with Smalltalk. OOP really took

off in the mid 1980s with C++—some argue because it was well suited to make GUIs, which were booming in popularity. OOP features were then added to several languages, such as Perl, Ada, BASIC, Lisp, and Pascal, and several new languages that embraced the OOP methodology were developed, like Java and Eiffel.

The main idea behind structured programming is to divide and conquer. As computers, technology, and software have advanced, programs have become larger and more difficult to write and maintain. Structured programming breaks down complex programs into simple tasks. The rule of thumb is that if a task is too complex to be described simply, then the task needs to be broken down further. When the task is small enough to be self contained and easily understood, then the task can be programmed.

Structured programming gave rise to a number of other movements, Object-Oriented Programming being one of the more important ones. A number of languages in the 1980s begin to pick up OOP features. In 1987 Apple creates a language called HyperTalk, used to script Hypercard stacks. This preempted the release of Perl in 1988, a still popular higher-level language that combined popular aspects of C, SED, AWK, and CSH (see Figure 1.4).

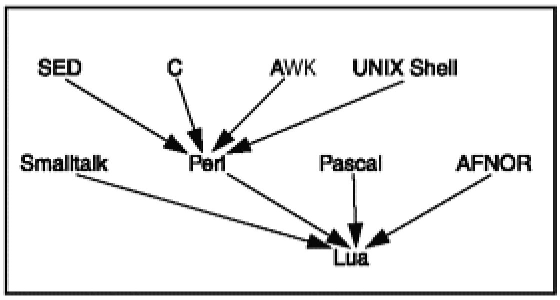
Figure 1.4. The big picture, high-level language family tree



Introducing Python

Python is a high-level, interpreted language originally intended for prototyping or as an extension language for C applications. The language is considered to be an interactive, object oriented-scripting language. It was designed to be highly readable, uses English keywords frequently where other languages use punctuation, and has fewer syntactical constructions than other languages (some call this clear syntax). Python's history is outlined in Figure 1.5.

Figure 1.5. The Python language family tree



Python is renown for its use of white space, as it uses space to delimit program statements. The language takes a lot of features from ABC, a language designed with beginners in mind, so Python is a great beginning language. Python supports the development of a wide range of applications, from simple text processing to WWW browsers to games (as we will shortly see).

Python Features

Python was developed by Guido van Rossum at the National Research Institute for Mathematics and Computer Science (otherwise known as CWI) in the Netherlands. Python is copyrighted, but the source code is open source and freely available. And yes, the language is named after the TV series Monty Python's Flying Circus.

Python's feature highlights include:

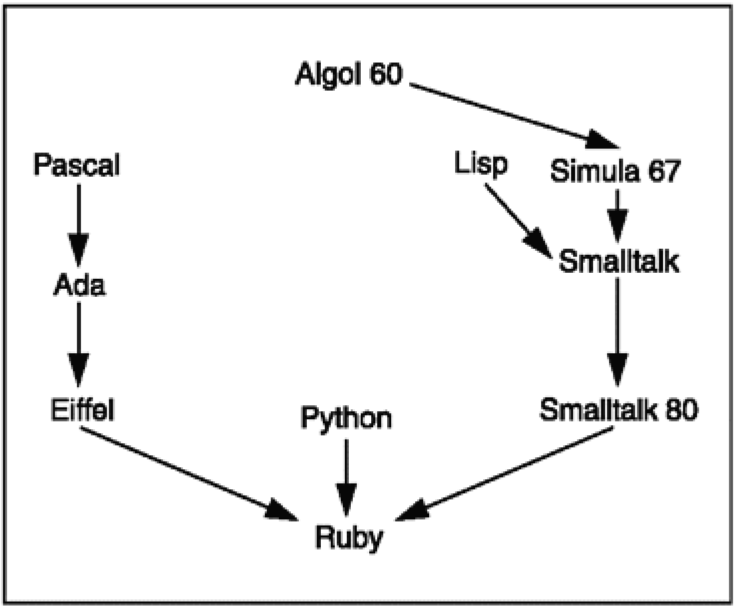
- A broad standard library, one of Python's greatest strengths. The bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh. The library contains built-in modules (written in C) that provide access to system functionality (for instance, file I/O) that would normally be inaccessible to a high-level language. Standard libraries include files, strings, math, threads, sockets, CGI, HTTP, and FTP.
- Support for an interactive mode in which you can enter results from a terminal right to the language, allowing interactive testing and debugging of snippets of code.
- An extensive graphics package.
- It is very portable, with interpreters for most operating systems.
- Support for OOP in the form of multiple inheritance, classes, namespaces, modules, objects, exceptions, and late (runtime) binding.

- Support for functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- Very high-level dynamic data types.
- Dynamic type checking.
- Automatic garbage collection.
- Run type checking.
- It is easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Introducing Lua

Lua is a byte-code interpreted glue language with extensible semantics as a primary feature. Lua is considered lightweight and was designed for extending applications. Its predecessors are Smalltalk, Perl, Pascal, and AFNOR, as illustrated in Figure 1.6. Lua is considered an excellent language for rapid prototyping and scripting and is implemented in C.

Figure 1.6. The Lua language family tree



Lua Features

Lua was developed at TeCGraf, the computer graphics technology group at the Pontifical Catholic University of Rio de Janeiro in Brazil. The team credited with developing the language in 1994 includes Waldermar Celes, Roberto Ierusalimschy, and Luiz Henrique de Figueiredo. The language qualifies as open source but it is not in the public domain, and Tecgraf holds the copyright. Lua means moon in Portuguese.

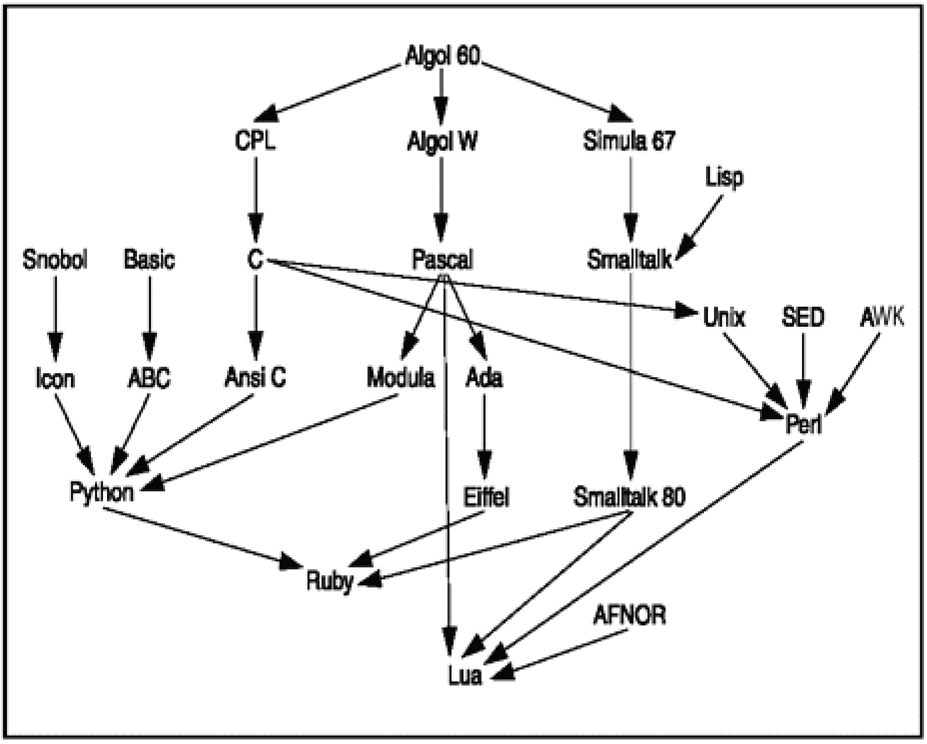
Lua feature highlights include:

- A simple Pascal-like syntax.
- It is dynamically typed.
- Automatic memory management and garbage collection,
- Powerful data description constructs like associative arrays.
- OOP mechanisms such as classes and inheritance.
- User-controlled type constructors.
- Fallbacks for extending the meaning of the language in unconventional ways.
- Its programs are compiled into byte-code and then interpreted, simulating a virtual machine.

Introducing Ruby

Ruby is considered a pure, modern, object-oriented language. Figure 1.7 shows how Ruby combined elements of Smalltalk and Eiffel. It sports a simple syntax inspired by Perl and Ada and is considered very readable, easy to maintainable, and clean, with only a few special syntactical situations. Ruby is highly portable and runs on UNIX, Max, Windows, DOS, OSX, and Amiga platforms.

Figure 1.7. The Ruby language family tree



Ruby Features

Ruby was created by Yukihiro Matsumoto in 1993. The language is open source, and its use is covered under the GPL artistic license. Matz, as he is affectionately known, knew Python, but he didn't like it because it wasn't pure OOP. He wanted a genuine OOP scripting language that was easy to use and write. Ruby's name, however, is a takeoff on Perl and is named after a colleague's birthstone.

Ruby feature highlights include:

- Pure OOP. Every bit of data in Ruby is an object, even basic types. There are no Ruby functions, only method calls (every function is a method). Unified class/type hierarchy, metaclasses, and the ability to subclass everything. There is also only single inheritance.
- Dynamic loading.
- Exception handling.
- Automatic garbage collection.

Summary

You should now feel pretty comfortable describing a high-level language, and you probably know enough about Ruby, Python, and Lua to be a source of interesting conversation at a local coffee house. You might be able to name a few predecessors of each language and have an idea of how each is related (check out the big family tree in Figure 1.4 to put this history in perspective). You should definitely understand what a scripting language, interpreter, and compiler are before you go onto the next section. If you can also pull facts about OOP and open source, give yourself an A and move on to Chapter 2.

Important points from this chapter:

- Languages possess a syntax that defines the order, arrangement, and structure of the system of communication.
- All computers CPUs have an internal machine language that they execute directly.
- All data in a modern digital computer is stored as binary on and off states. The tools used to manipulate these on/off states are coded in a numerical representation, normally consisting of two pieces of information: operation codes and addresses.
- Assembly language is one step higher than machine language and consists of numeric instructions for specific computer architecture.
- High-level languages act as translators between programmers and low-level computer instructions and closely resemble everyday human language, making them much easier to learn than their low-level equivalents.
- Interpreted languages translate code step-by-step during runtime.
- Compiled languages translate code before a program is run in a process called compiling that turns written code into a runable executable or runable byte-code.
- A scripting language is a high level language designed for "scripting" the operation of a computer.
- High-level languages save human time, low-level languages save computer time.

Questions and Answers

1: Q: Why would I want to program a game in Python, Ruby, or Perl if C is faster?

A: A: Speed is obviously essential for games, but most of the slowdown of a particular game engine occurs in only a few places. Many companies opt to do the bulk of their game development in a high-level language, and then delve deep into C or assembly for specific, processor-bogging graphics. Python and Ruby were designed with this mind, so they lend themselves well to extending themselves in C or any other language. Lua itself is created with C, and can also work with that family quite easily.

2: Q: What types of games are usually developed with these languages?

A: A: Python has been the engine behind a number of titles but is widely known for allowing companies to easily and quickly create graphically rich, Myst-like worlds, and cartoon-animated games like the award-winning titles from Humongous. It is also the glue behind a few major motion picture CGI shops, used in various ways for computer graphics production. Lua has been a hidden secret of game companies for a decade and has been the scripting agent behind a number of popular games on platforms ranging from handhelds to PCs to the Xbox. Normally Lua is used for game scripting, and not the game engines themselves. Ruby is still gaining in popularity, and many of its larger game projects are still in development. Until recently, Ruby was regarded mostly as an all-purpose OOP language, and much of its development thrust has been in enterprise-level Internet applications. Ruby is capable, however, of the same sorts of game development that Python is and has a few extremely strong graphics and sound toolkits and libraries.

3: Q: Why is it easier to find projects with Python and Lua than with Ruby?

A: A: Ruby is just as pervasive as the other two languages, except that the bulk of development and documentation is happening in Japan. Ruby enthusiasts claim that the language is much more popular than Python in Japan, which is evident by the growing number of Ruby books that are available in Japanese.

Exercises

1: Answer the following as True or False:

A. High-level languages are difficult to port to other architectures.
B. High-level languages are called high-level because they resemble human languages.
C. Programming languages are translation systems.
D. The biggest problem with low-level languages is adapting them to different platforms.

2: Fill in the blanks in the following sentences:

A. A computer program that converts assembly language to machine language is called a(n) _____.
B. A computer program that translates code during the program _____ is called an interpreter.
C. An example of a high-level language besides Python, Ruby, or Lua is____ (give at least two examples).

3: What is the only language that a computer can understand directly?

4: Imagine your ideal programming language. Make a list of ten must-have features that your perfect programming language would possess.

5: Describe the differences between high-level, interpreted, and scripting language features (I warned you there was a quiz coming up)!

Chapter 2. Python, Lua, and Ruby Language Features

The limits of my language means the limits of my world.

——Ludwig Wittgenstein

This chapter serves as an introduction to common features of Python, Lua, and Ruby. I introduced each language in Chapter 1, and in this chapter I'll be going into more details of the languages.

There are two main goals for the chapter. The first is to give you a foundation for learning how to use these languages by covering a few features they have in common. The second objective for this chapter is to start you coding.

These objectives are met in the main sections within this chapter. The first section covers a few of the common base programming commands the languages have in common. The second section walks you through a "Hello World" sample in each language.

Syntactical Similarities of Python, Lua, and Ruby

One great bonus to learning similar languages at once is the overarching familiarity that comes with common elements. All programming languages have some similar features, and these three languages in particular are based on similar premises and ideas. This makes it possible to share the learning curve, so to speak. Python, Lua, and Ruby share the following particularly important programming elements:

- Comments and commenting
- Math and algebraic functions
- Variables
- Lists and strings
- Program structure

Comments and Commenting

All modern programming languages allow programmers to insert comments into their code. Comments are extremely important, not only to the professional who needs to write code that other people may need to change or maintain, but also to individuals or independent programmers who need to look at their code again at some point in the future to see how they did something or modify an existing program.

Most languages reserve the use of the pound sign (`#`) to designate a one-line comment. You will find the `#` symbol used in this way in AWK, Perl, PHP, and C, but most importantly for us, in both Python and Ruby. Here is an example of commenting in Python and Ruby:

```
PYTHON
RUBY
# This code sample has only comments
# The computer, compiler, or interpreter will for the most part
# Ignore all of these lines
# Simply because they start with a pound sign
```

Lua has its very own comment designator: two dashes in a row (--). Here is an example:

```
LUA
-- This code sample has only comments
-- The computer, compiler, or interpreter will for the most part
-- Ignore all of these lines
-- Simply because they start with dashes
```

Math and Algebraic Functions

When it comes right down to it, your computer is speaking a language of 0s and 1s. It's no surprise, then, that math tools, functions, and operators tend to be similar across all languages. You can pretty much bank on functions like add (`+`), subtract (`-`), multiply (`*`), and divide (`/`) being available no matter what programming language you are using.

Another commonality between Python, Lua, and Ruby is using parentheses `()` to state precedence; this comes right out of high school algebra. For instance, the answer to this code example will be different depending on the order of operations: 1+2*3 = X.

If you perform the operations from left to right, X will equal 9, but if you do it from right to left, X will equal 7. Python, Lua, and Ruby (along with many other languages) use parentheses to specify the order in which computations should be performed if you wish to override the natural order of operations. If you needed to specify that you multiply before adding in the following example, you can use parentheses around the multiplication, forcing the multiplication to be computed before the addition:

```
1+ ( 2*3 )  = X
```

Parentheses are often used with other programming structures to perform comparisons and to make decisions during the program flow. Understanding how to pose and evaluate comparisons is a crucial skill for any programmer or computer scientist. Because they are so often used, many different types of comparisons have been developed.

Boolean Logic

A mathematician named George Boole invented Boolean algebra in the nineteenth century. Boolean Algebra only has two values: True and False (this is sometimes called two-valued logic). It may be difficult to balance your checkbook with Boolean algebra, but it's extremely easy to create decision and logic trees with it.

Boolean expressions often involve comparison operators to help evaluate truth or falsehood. Operators such as equal (=), less than (<), and greater than (>) should look familiar to you if you didn't skip your high school math classes. These constructs are so common and useful that many languages use them. Python, Lua, and Ruby all use the same comparison symbols, illustrated in Table 2.1.

Comparison operators are normally used to form expressions that can be evaluated as True or False. For example:

```
1 < 2 - Evaluates to TRUE
1 > 2 - Evaluates to FALSE
1 = 2 - Evaluates to FALSE
```

Sometimes you need to make comparisons in groups. A program may need to ask, "Is player character one an Elf AND a Wizard?"

```
Player1 = (Elf AND Wizard)
```

Typical comparisons use logical structures: logical AND, logical inclusive OR, and logical NOT. Logical AND along with logical OR are used to combine conditions or statements. Lua and Python try to keep the constructs simple for the reader by using common English words, as do most high-level languages, including Eiffel, Ada,

Smalltalk, Lisp, and Perl. You can designate logical AND, OR, and NOT by using the command words `and`, `or`, and `not`, respectively.

Ruby takes a slightly different course and follows convention, using the same programming symbols that the popular C family (C, C++, and C#) uses to designate AND, OR, and NOT: &&, ||, and !. These differences are also illustrated in Table 2.1.

The logic constructs AND, OR, and NOT are normally used with Boolean True and False to form simple and complex programs. These constructs are sometimes called Boolean operators.

Boolean operators are evaluated differently when in combination with each operator. For the AND operator, the combination of two True values results in True; all other combinations evaluate to False, as illustrated in Table 2.2.

Table 2.2. Boolean AND

Operators	**Evaluation**
True AND True	Evaluates to True
True AND False	Evaluates to False
False AND True	Evaluates to False
False AND False	Evaluates to False

For the OR operator, as long as one of the values is True, then the expression evaluates to True, as shown in Table 2.3.

Table 2.3. Boolean OR

Operators	**Evaluation**
True OR True	Evaluates to True
True OR False	Evaluates to True
False OR True	Evaluates to True
False OR False	Evaluates to False

The NOT operator is called the complementary operator. It reverses the truth-value, as shown in Table 2.4.

Table 2.4. Boolean NOT

Operators	**Evaluation**
NOT True	Evaluates to False

Table 2.4. Boolean NOT

Operators	**Evaluation**
NOT False	Evaluates to True

Once you understand Boolean logic, comparison operators, and logical structures, you can create very complex decision trees, like this:

```
# The following line evaluates to Boolean FALSE
(((1+2)*5) =11) and ((5*6) != (7*6))
# The Following line evaluates to Boolean TRUE
((1+1) = 5) or ((5*6) = 40 and ((5/4) = 2*.5)) or ((50/5) = 10)
```

Table 2.1. Common Math Functions in Python, Lua, and Ruby

Function	**Python Command**	**Lua Command**	**Ruby Command**
Add	+	+	+
Subtract	-	-	-
Multiply	*	*	*
Divide	/	/	/
Equal (assignment)	=	=	=
Equal To	==	==	==
Less Than	<	<	<
Greater Than	>	>	>
Logical NOT or Not Equal To	not	not	!
Logical AND	and	and	&&
Logical OR	or	or	||
Square Root	sqrt	sqrt	sqrt
Exponent	exp	exp	exp
Absolute Value	abs	abs	abs
Basic Sin	sin	sin	sin
Cosin	cos	cos	cos
Tangent	tan	tan	tan
Logarithm	log	log	log
Truncate/Round	round	round	round
Floor	floor	floor	floor
Ceiling	ceil	ceil	ceil
Power	**	^	**

Variables

Computers and computer programs manipulate data. Variables are holders for data any computer or program might need to use or manipulate. Variables are usually given names so that a program can assign values to them and refer to them later symbolically. Typically a variable stores a value of a specific given type like:

- An integer or whole number
- A real or fractional number
- A character or a single letter of an alphabet
- A string or a collection of letters

Many languages need to know in advance what type a variable will be in order to store it appropriately. Since computers are finite in memory, there are often several different numerical designations, depending upon how big a number can grow or how that number needs to be represented in binary.

Others languages are more flexible in dealing with variables; this is called dynamic typing, as I mentioned in Chapter 1, and is a common high-level language feature. Even with dynamic typing, most programmers declare variables at the start of their program out of convention. This consists normally of dreaming up a name and then declaring a data type for the variable. Typical variable types are listed and described in Table 2.5.

Table 2.5. Typical Variable Types

Variable type	Description
Boolean	Holds only True or False
Float	A number with a decimal point (floating decimal point)
Integer	A whole number
Null	No value
String	Ordered sequence of characters

Each language's variable types and how to use them are explained in more detail in their respective sections in this book, but there are a few commonalities I will mention here. For instance, null values are symbolized by `nil` in both Lua and Ruby, while Python uses the designation `none`. Both `nil` and `none` are treated as false in a Boolean sense.

```
PYTHON
# This assigns x a null or Boolean false value in Python
X = none
RUBY
LUA
# This assigns x a null or Boolean false value in Ruby or Lua
X = nil
```

A second example of similarity with variables is that Python and Ruby both use the `value` method to grab the value of a variable.

```
PYTHON
RUBY
# this code snip uses the value method to
return the value of x
x = 4
# This line grabs x and prints it in Python
print x.value
# This line grabs x and prints it in Ruby
$stdout.print(x.value)
```

NOTE

CAUTION

Many popular languages—for instance C, C++, and Perl—also use zero (0) as Boolean false. This is not necessarily the case in High-Level Land. For instance, in Ruby, anything not designated as `nil` or `false` is automatically true in the Boolean sense, even the number 0. This switch sometimes tricks converts from other languages.

Although similar in some ways, Python, Lua, and Ruby differ significantly in how they handle variables and types. They each follow slightly different paradigms that create differences on a basic level. These differences will become apparent as you delve into each language in the chapters that follow.

Lists and Strings

Lists are used to group things together. They are data structures designed to make life easier for the programmer. A list is simply a row of variables or data elements. They can be composed numbers, letters, or even constructs such as arrays, hashes, or even other lists. Lists are created in Python and Ruby by using brackets `[]`:

```
PYTHON
RUBY
#To create a list called lista with the numbers 1 through 10, just put
them in brackets
and separate them with commas:
lista = [1,2,3,4,5,6,7,8,9,10]
```

You can use the `+` symbol in each language to concatenate lists together:

```
PYTHON
RUBY
# To combine fish and chips list a with list b
lista = [1,2,3,4,5]
listb = [6,7,8,9,10]
# Just add them together into a new list
newlist = lista+listb
```

Since not all languages have direct support for strings, one of the time-saving features that high-level programmers often enjoy is built-in string handling. Not only are there common commands for working with strings, the memory management of strings is usually handled automatically.

A strings is basically just a list of characters. To get Lua, Python, or Ruby to recognize a string verbatim, you can place it between single parentheses, like so:

```
PYTHON
LUA
RUBY
# Python, Lua, and Ruby will recognize this as a string
'Enclose strings like this in single quotes'
```

You can also use math functions to make string comparisons in Python and Ruby, just like you can with lists. For instance, the + sign can be used for string concatenation with Python or Ruby, like so:

```
PYTHON
RUBY
# to combine the strings 'fish', 'and', 'chips'
stringa = 'fish'
stringb = 'and'
stringc = 'chips'
stringd = stringa+stringb+stringc
```

You will find that equal to (==) and not equal to (!=) are often used to compare different strings as well:

```
PYTHON
RUBY
# Is the password 'Enter' ?
# First, get the password
If password = 'enter'
        # Then you shall pass

If password != 'enter'
        # Then no such luck
```

Arrays

Arrays are similar to lists. They are both used for storing items or lists of items, but they keep track of the items in different ways. Arrays organize lists of items by a numeric index, an extremely powerful tool in programming.

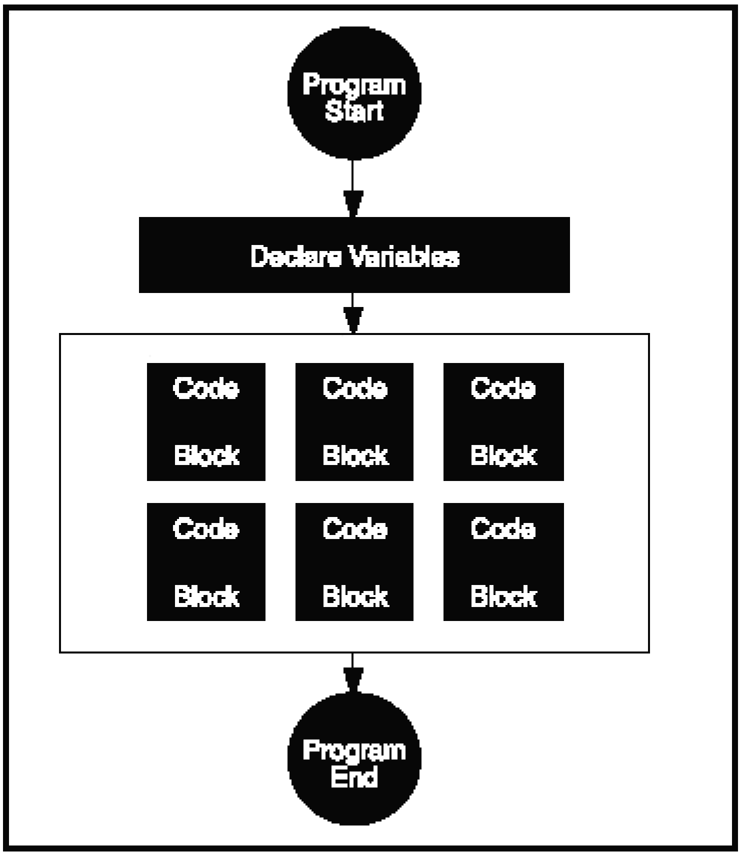
Although each of these languages handles lists in a similar way, they have somewhat different approaches for arrays. Ruby has a built-in array method, but, strictly speaking, Lua does not have built-in arrays and substitutes for them with table structures. Python has its own version of arrays called sequences.

Despite the differences, these languages handle arrays in similar ways. An example is the `sort` method or command. Ruby uses `sort` to put in order items within a hash or array, Lua uses `sort` to order a table, and Python uses `sort` to order a list. Similarities like these run deep through these languages, but can become confusing and difficult to wade through when switching between them frequently.

Program Structure

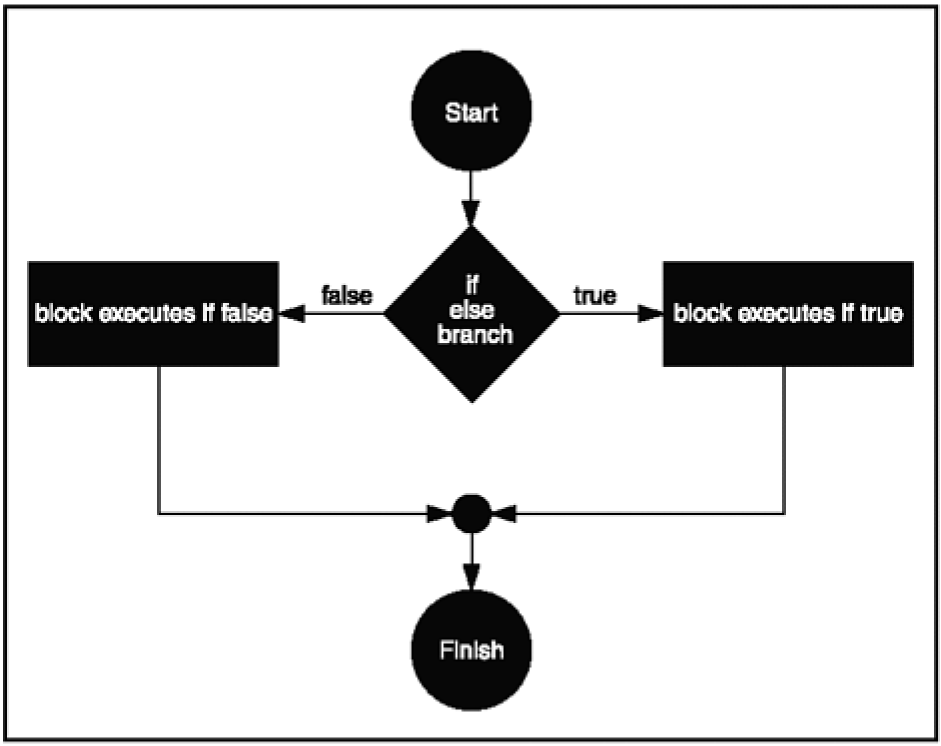
All programming languages have some sort of structure or flow to them. Most programs share a structure similar to that in Figure 2.1. Normally there is a statement that establishes the beginning of a program, then variables are declared, and then there are code blocks, which are also called program statements.

Figure 2.1. A typical program structure



Program statements provide the control of a program. They usually act as decision trees, executing different sections depending upon the input given. Figure 2.2 illustrates a structured-programming graph in which the ovals represent starting and ending points, the squares represent program blocks, and the diamond represents a decision to be made in the program that will send the flow down one of two branches.

Figure 2.2. A structured-program flowchart



Program statements come in a couple of different forms. So far in this book, I've used mostly simple statements. Simple statements are short expressions that perform specific actions. There are also compound or complex statements that generally consist of more than one line of code and use many expressions.

Statements that control which sections of code are to be executed are called control statements (surprise!) and consist of a few basic types.

- **Linear Control Statements.** Control is based on a logical sequence, and code is executed in the default order as it's listed in the source file.
- **Conditional Control Statements.** A condition is set that makes a decision on which block of code is to be executed.
- **Iterative Control Statements.** Blocks of code may be executed more than once in loops.

Linear Control Statements

Linear control statements are the most intuitive of type of program structure. In linear control, commands are executed in a sequential, ordered, linear manner. This usually equates to running one line at a time, like so:

```
Start Program
Run Command 1
Run Command2
Run Command3
End Program
```

Since English-speaking humans are most comfortable reading from left to right and from the top down, the same conventions are used in linear control.

Conditional Control Statements

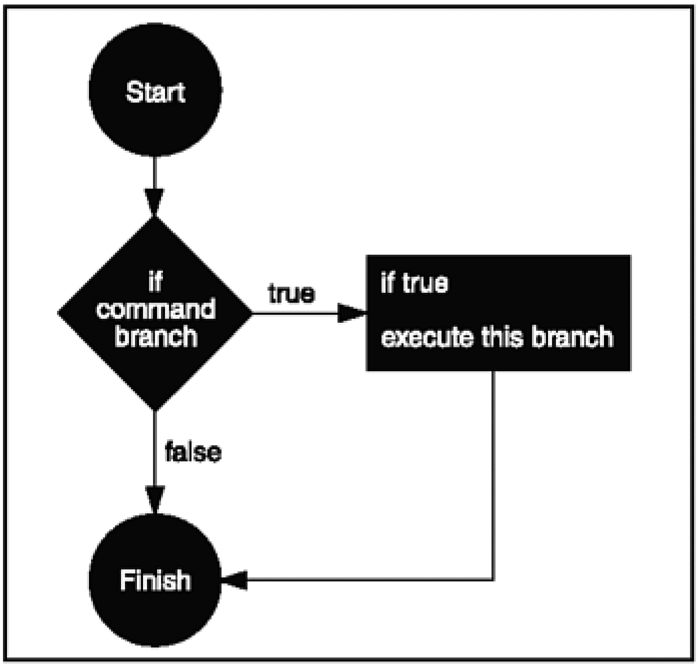
Statements that are considered conditional are often referred to as if/else statements. The commands `if` and `else` determine which lines or blocks of code might or might not run, depending on the flow of the program. Programmers generally use these as branches to initiate actions that are dependent on user input.

The `if` command is the foundation of all conditional statements. `if` checks a specified condition for truth value. If the condition is true, then `if` executes a code block that follows. If the condition is not true, the code block is skipped.

```
if (this condition is true)
(then this happens)
```

Figure 2.3 depicts the flow of a program going through an `if` statement. The flow goes through the diamond branch, which executes the code block (the square) if the condition is true or continues to the ending oval if the condition is false.

Figure 2.3. A generic example of a program flowing through an `if` statement



Python uses `if` for command flow in the following way:

```
PYTHON
# Examples of and if statement checking the truth of X being greater
than 90 in Python
if X > 90
```

```
        Then do this
```

Ruby and Lua are similar, but use an `end` command to designate the end of an `if` structure.

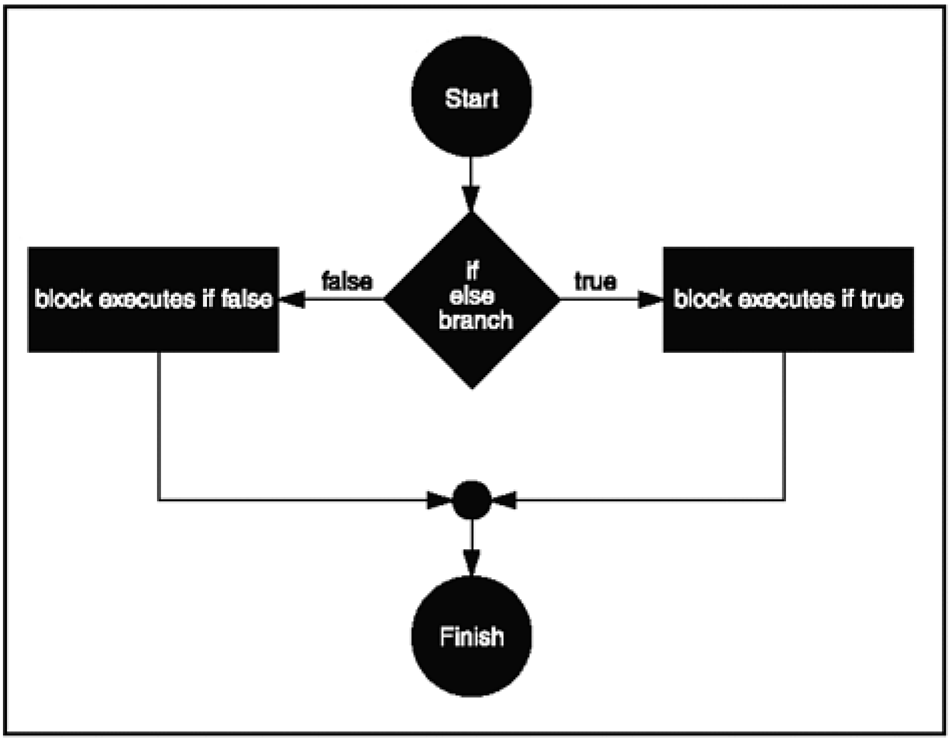
```
RUBY
LUA
if X > 90
then do this
end
```

The `else` command is another common conditional that can follow an `if` statement. When an `if` statement returns a value of false, the code block held by `else` executes. This creates a fork in the program, where either the `if` block or the `else` block is executed. When using `else` and `if` together in Python, Ruby, or Lua, the general syntax looks something like the following:

```
if (this condition is true)
then this happens else (this happens instead)
```

This series of `if` and `else` statements allows code to make decisions based on variables or input. When the program flow has two possible execution choices, it is known in structured programming as a double selection. The `if/else` statement is illustrated in Figure 2.4.

Figure 2.4. The top-down flow of an `if/else` statement. The false and true branches both execute blocks of code



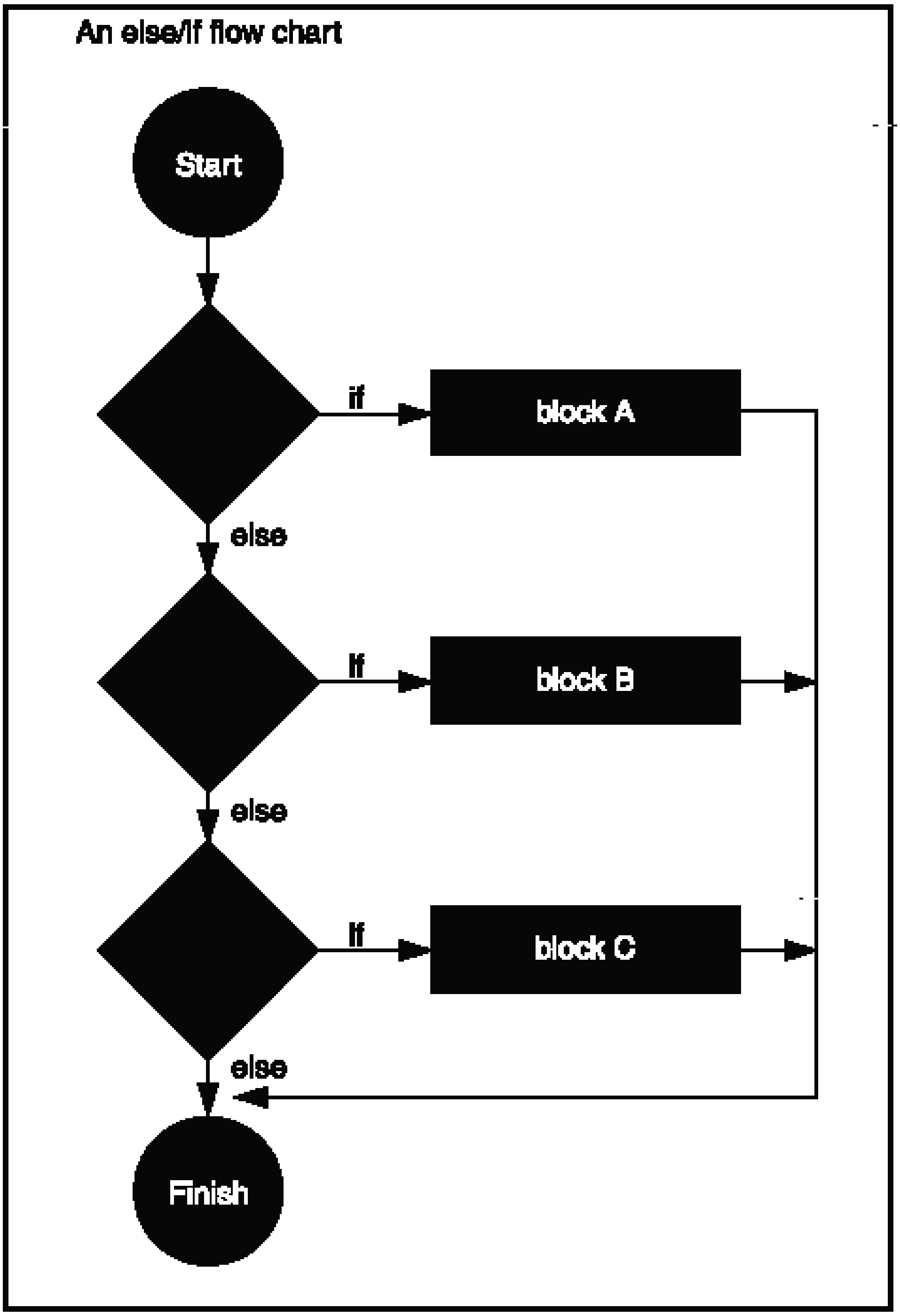
A double selection can be limiting because there are only two forks that the program can take. If you need to program for multiple paths, you can use the `elsif` command in Ruby, the `elseif` command Lua, or the `elif` command in Python, all of which are equivalent.

You can use multiple `elsif/elif` statements in a row to create one long string of conditions for which to check. However, only one `else` statement can follow an `if`. The syntax for these statements looks like the following:

```
if (this first condition is true)
(then this first program block runs)
elsif/elseif/elif (this second condition is true)
(then this second program block executes)
elsif/elseif/elif (this third condition is true)
(then this third program block runs)
else
(this fourth block fires instead)
```

As we get deeper into each language, each will start to have its own distinct flavor, and they will begin to appear different. You can see how different in the following code, which displays the typical `elsif/elif/elseif` flow in each language. Figure 2.5 also shows a typical `elsif/elif/elseif` program structure.

Figure 2.5. Structure of a program illustrating multiple `else/if` branches




```
PYTHON
# Example of elif In Python
If X > 90
        print "this"
elif X < 90
        Print "this instead"
Else:
        Print "this"
LUA
# Example of elseif In Lua
if x>90 then blocka {elseif x<90 then blockb} [else blockc] end
RUBY
```

```
# Example of elsif in Ruby
if x > 90
        then this blocka fires
elsif x < 90
        then this blockb fires
else
        blockc fires
end
```

Iterative Control Statements

Programming languages must have a facility to allow sections of code to be repeated, or iterated. Iteration is possibly a computer's greatest strength. There are several variations of constructs that are used to iterate program blocks; these are commonly called loops.

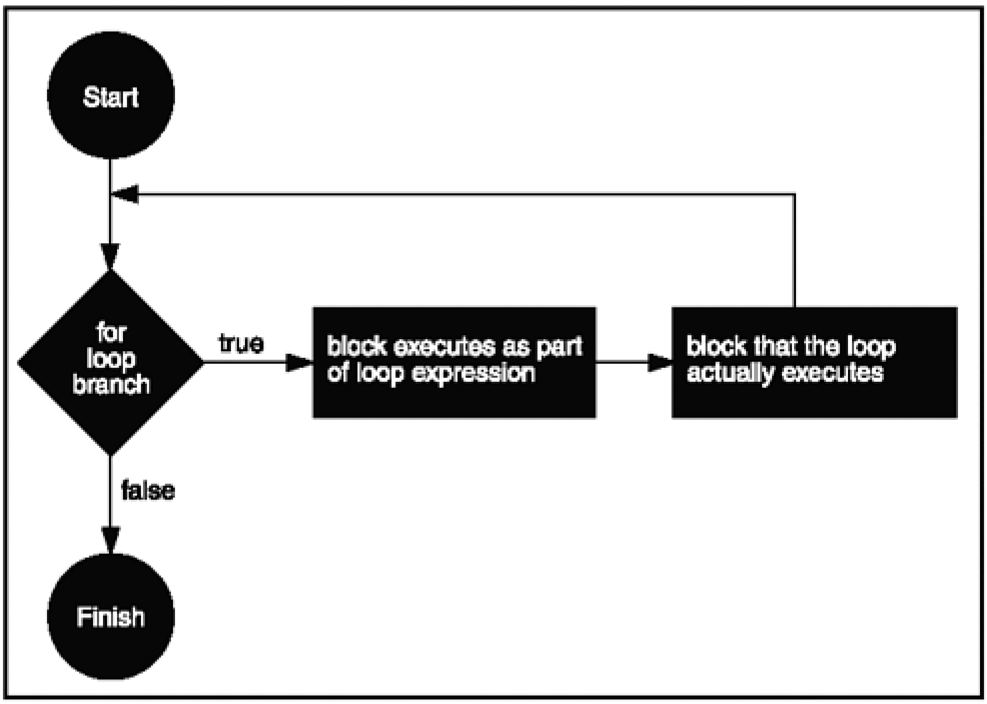
The for Loop

The `for` loop is probably the most common loop in programming. It takes a few separate conditions to execute and takes on the following general structure:

```
for (the length of this expression)
(execute this code block)
```

Figure 2.6 shows the structured program flow of a `for` loop. Notice that there are two programming blocks: the first is the code that executes as part of the loop expression, and the second is the block that the loop executes.

Figure 2.6. The flow of a `for` loop



Although iteration commonly is needed when programming, and a built-in `for` construct exists for each of these languages, each has its own peculiarities.

Python's `for` loop uses a counter and a range to determine how many times to loop a given code block. The counter is incremented each iteration of the loop until the counter reaches the end of the range and the loop is complete.

```
PYTHON
for counter in range ( X ):
        block
To create a for loop in Python that loops 10 times, do the following:
PYTHON
 for counter in range ( 10 ):
        block
```

Lua's `for` statement works in the same principal way but has two forms, one for numbers and one for tables. The numerical `for` loop has the following syntax:

```
LUA
for name '=' exp1 ',' exp2 [',' exp3] do block end
```

The first expression (`exp1`) is the counter, the second (`exp2`) is the range, and the third (`exp3`) is the step (the step is automatically a step of 1 if omitted). Therefore, a `for` loop in Lua that would run a block 10 times would look something like the following:

```
LUA
for name = 1 ,10, 1 do block end
```

Ruby has a unique way of dealing with `for` loops, and iterators in general. Ruby uses a number of predefined classes with built-in methods to provide iteration functionality. There are several ways to accomplish the same 10-iteration loop in Ruby:

```
RUBY
10.times do
        block
end
```

Or

```
RUBY
1.upto(10) do
        block
end
```

Ruby also has a comparable `for/in` construct with a similar structure:

```
RUBY
for i in 1..10
        block
```

```
end
```

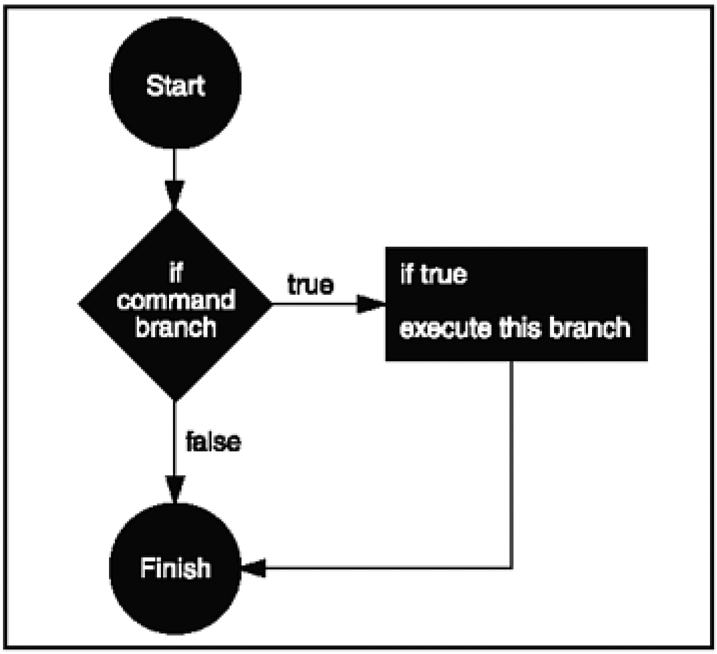
and a built in `loop` iterator that looks this:

```
RUBY
i=0
loop do
        i += 1
        next if i < 3
        block
break if i > 4
end
```

The while Loop

A second common loop is known as the `while` loop (sometimes known as the `do/while` loop). A `while` loop is normally used to keep a section of code continually running while a certain condition is true. The flow of this loop is shown in Figure 2.7.

Figure 2.7. A flowchart that illustrates a typical `while` loop



The `while` loop takes on the general structure of:

```
while(this statement is true)
        do (execute this block)
```

Each language, again, has its own nuances, but the `while` loop looks fairly similar in each.

Python's `while` loop is almost identical to the `for` loop:

```
PYTHON
X = 100
while X < 100:
        block
```

Note that these examples could execute a never-ending loop unless a way to increase x was added.

Lua's `while` is almost identical to Python's, but with substitution of parentheses for the end colon and the addition of an `end`:

```
LUA
while ( X> 100) do
block
end
```

Ruby's `while` is also almost identical to Python's:

```
RUBY
while X < 100
        block
end
```

Miscellaneous Similarities

As you read through this book, you will find more and more similarities between the languages. In addition to commenting, mathematics, lists, variables, and program structure, there are a number of other significant similarities. Some of these I will point out as the book progresses, and others you'll discover on your own. A few of the more significant ones are illustrated in this section. Table 2.6 lists a few miscellaneous commands that have similar or the same names.

Table 2.6. Similarly Named Commands

Function	**Python Command**	**Lua Command**	**Ruby Command**
Access read/write	`a[e]`	`a[e]`	`a[e]`
Runtime evaluation	`eval`	`dostring`	`eval`
Duplicate n times (string repeat)	`*`	`strrep`	`*`
ascii to character	`chr`	`strchar`	`chr`
Value	`v`	`v`	`v`

End-of-Line Characters

Knowing where a command line ends is important for understanding program flow. One line of code is usually over at the end of the line, when a return is entered. The end of a line may also end in an "End of line" command such as a colon (): or semicolon (;). Python and Ruby both use the semicolon symbol as an end of line command to end a statement; in Lua the semicolon is optional. In Python and Ruby you can also simply use an end of line character (or a return).

```
#  Sample of an end of the line statement
This code line ends at the semicolon;
This is a second, separate line of code;
```

Breaking up a line is useful if the line is too long and you need to go on to the next line. Both Ruby and Python both use the \ (backward slash) to signify that the command goes on to the next line.

```
PYTHON
RUBY
# Sample of using a \ to extend a line of code
This code line ends at the semi colon;
This snippet goes on to the next line\
And ends here
```

OOP Structure

Since each of these languages is object oriented to some degree, and they are all based on similar strategies, it follows that they possess similar object-oriented constructs. This is especially true for Python and Ruby, whose commands for method invocation, class declaration, and scope are identical. In fact, method invocation (and scope) uses a very recognizable structure for OOP veterans:

```
object.method(parameter)
```

As you can see, the `.` operator is used to define scope as well as a record selector. The command class is also used to designate a class in both languages.

Function Calls

All three languages have similar commands for function calls, the typical syntax being:

```
function(parameters)
```

In Ruby, you can call a function without any parameters just by naming it:

```
function
```

In Lua and Python, you must still specify that there are no parameters with parentheses:

```
function()
```

The command `return` is used by Ruby and Python to break the function control flow and return a value.

Hello World Samples

Now that you've seen snippets and small samples of code, it's time to look at what a fully functioning program looks like in each language.

Programming in each language is explained in depth in each of the following sections, so don't be concerned if the code sample that follows appears foreign. This is just a sample to whet your appetite.

The Python Environment

Python is at home in a number of different environments and can be programmed via command line, script, or debugger. This section will help you install Python on your system and will demonstrate the different options available when you need to sit down and write code.

Installing Python

This book's CD-ROM comes with the tar archive and the Windows installer for Python Version 2.2.2 (released in October of 2002); you can find them in the Python folder. You can also download Python installers for a number of other different platforms from the Python.org Website at http://www.Python.org/download.

As of this writing, Python 2.32 alpha is available from Python.org in Windows. The alpha is also included on the CD, but the samples in this book were written with Version 2.22.

Simply double-clicking on the Python-2.2.2.exe file located on the CD under \PYTHON will run the Windows installer. The installation is fairly straightforward; just click OK on the windows that pop up (see Figure 2.8).

Figure 2.8. The Windows Python Installer in action



If you'll be installing for a UNIX platform, you'll need to perform the regular steps for unzipping the tar archive and installing (`gunzip`,`tar`, `./configure`, `make`, and `make install`). You will want to perform this action as `root`.

If you'll be installing Python on a Macintosh, you will want to use the MacPython222Full.bin file for OS 8.6 or higher—except for OS X. For OS X, you will want to use the standard tar archive. If you are running Mac OS X 10.2 or later, Python actually ships on the platform, and you won't need to install it at all. Python for the Macintosh is maintained by an independent programmer named Jack Jenson. You can find patches for a few older Mac operating systems and more information at his Website at http://www.cwi.nl/~jack/macpython.html.

If you are running Red Hat and want to grab the RPM sources instead of using the tar, they are available for some distributions from Python.org. Just go to the Website and check out the Download page.

If you do decide to use a version of Python other than 2.2.2, be sure you use Version 2.0 or higher. Python went through a few significant changes from Version 1 to Version 2, and if you use a version earlier than 2.0, you may have trouble running the code samples in this book.

The Python language is copyrighted by Stichting Mathematisch Centrum in Amsterdam. However, it is free to use, copy, modify, and distribute and is OSI (Open Source Initiative) certified. You can find a copy of the license and copyright in the Python folder on the accompanying CD, and again in the Licenses folder.

NOTE

CAUTION

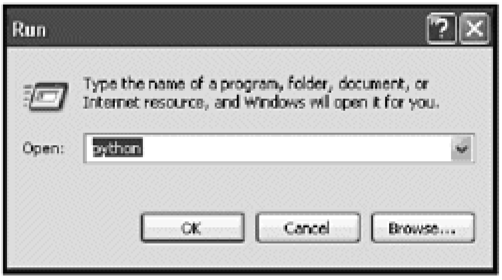
Intellectual-property attorneys exist for a reason: It is possible to get in legal trouble selling open-source software. Luckily, the open source community is fairly watchful about intellectual property law, and licenses are becoming somewhat standard and easier to read. There are risks associated with incorporating open source code into commercial endeavors that should not be taken lightly but these risks should not prevent you or your company from using this viable and effective resource. If you have concerns or questions about a license, or about using any open source software in a major enterprise, by all means ask an expert.

Running the Python Interpreter

After you've installed Python on a Windows machine, the Python interpreter is accessible via the `run` command. Simply do the following:

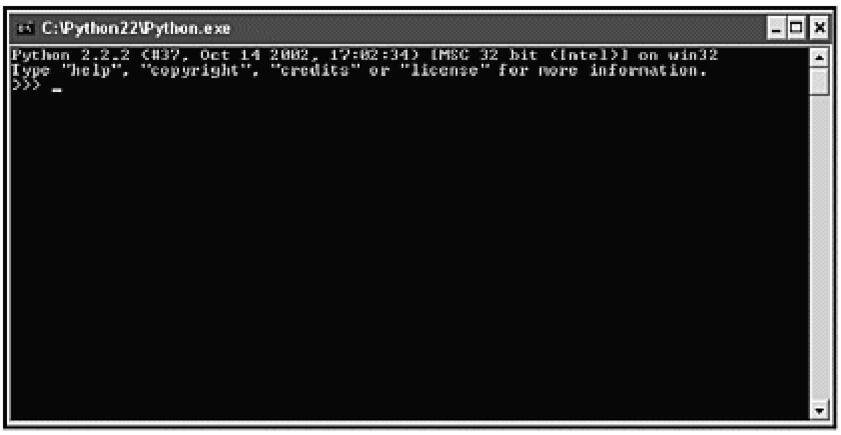
1. Open your Start menu.
2. Select Run.
3. Type python and hit OK, as illustrated in Figure 2.9.

Figure 2.9. Windows XP waits for a command from the user to launch Python



You will get a command window that looks like Figure 2.10 saying:

Figure 2.10. The Python interpreter awaits your command



```
Python 2.2.2 <#37, Oct 14 2002, 17:02:34> [MSC 32 bit <Intel>] on
win32
Type "help", "copyright", "credits" or "license" for more information
>>>
```

NOTE

CAUTION

The Python installer tries in good faith to set your machine path variables so that you can run the Python binaries from the command line or anywhere else for that matter, but the installer may not be able to on your particular platform. If you cannot get Python to launch from the command line, you may have to set the path variables yourself or simply run the Python (command-line) entry that is added to the program files listing under Python 2.2.

On a UNIX system, the Python interpreter is usually installed as /usr/local/bin/python, but of course where the interpreter lives is an installation option left up to you. You will need to put /usr/local/bin in your UNIX shell's search path to make it possible to start the interpreter by typing the command `python` to the shell.

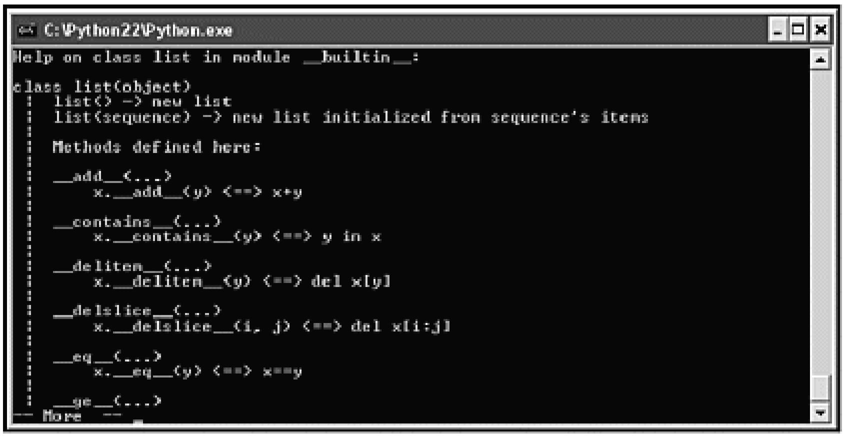
The Python interpreter actually operates somewhat like a UNIX shell—it reads and executes commands interactively. The interpreter can also be called with a filename argument or with a file as standard input.

Go ahead and test out the interpreter. You can start by executing various one-liners like `print "hello world"` or `5*5`. The interpreter is great for testing out certain functions. The interactive help is also very useful. Type the following at the interpreter's prompt:

```
>>> help (list)
```

You will receive information on the `list` command, its syntax, and samples of its use (as illustrated in Figure 2.11).

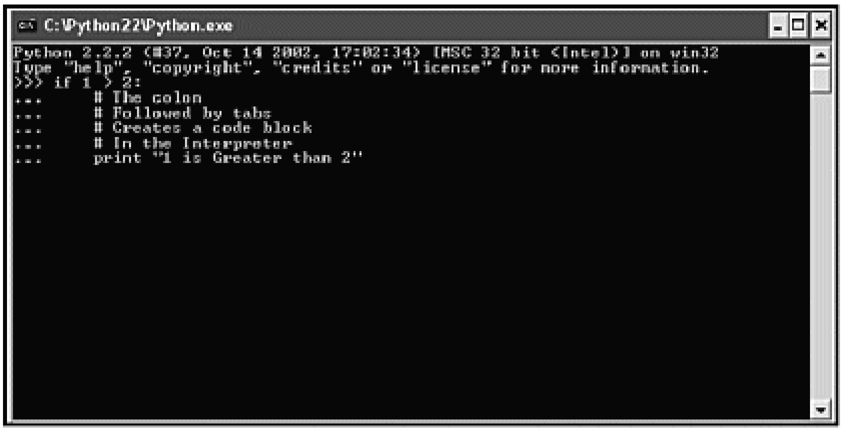
Figure 2.11. Python's interpreter shows how to use the built-in `list` class object



In this mode, which is called interactive mode, you can type any Python command and it will work just as if you typed it from a script (with a few differences). If you type a command that returns a value of some sort (except assignments), the interpreter will print the result automatically. This is great for experimenting and for testing a specific language feature when you need to get the syntax right. The interpreter isn't very helpful, however, when it comes to large sections of code or actual programs, which you will want to write and then execute at once.

What the interpreter is quite good at, though, is running through the code snippets and short examples you'll find in the next few chapters. You can easily run one-liners to test a particular Python feature, or you can write short, multiple-line code snips by first using a colon and then tabs to delineate a code block (as illustrated in Figure 2.12):

Figure 2.12. The Python interpreter is poised to run this five-line code snippet



To exit Python's interpreter, hit Ctrl-Z on Windows and then press the Enter key, or in UNIX, hit Ctrl-D.

NOTE

TIP

When a script file is used, it is sometimes useful to be able to run the script and enter into interactive mode afterwards. You can do this by passing the `-i` (i is short for "interactive") argument to the script.

NOTE

TIP

When you use Python interactively, you can set standard commands to execute every time the interpreter is started. You can do this by setting the environment variable `PYTHONSTARTUP` to the name of a file containing the commands (this is similar to the .profile feature in UNIX). This file is only read in interactive sessions, not when Python reads commands from a script.

Creating Python Program Files

You can also run Python programs from a file. The usual extension for a Python program is .py. To create new Python program file, just fire up your favorite text editor, type in a few commands, and save the program as something .py. For instance, on Windows, open up Notepad and type in:

```
print "Hello, World!"
```

and save the file as hello.py. You can then save the file to disk, open up your command line, browse to hello.py, and run it, as shown in Figure 2.13.

Figure 2.13. A Python program file runs on Windows



You can create Python program files on other operating systems just as easily, except that in a Posix environment (UNIX or Linux), you will need to include a line at the top of each file that points to where Python is installed on your system, like this:

```
#!/usr/local/bin/python
```

This makes the file directly executable, like any other shell script.

Python's "Hello World"

A Python "Hello World" looks like this:

```
#!/usr/bin/python
############
# HELLO_PYTHON_1.py
# This program displays the string "hello" .
# It first shows the path to python, then creates a short loop, and
then prints the
string.
############
while (1) :
         print "Hello!";
```

Most of this script is made up of comments. Python ignores lines that start with the # symbol, so coders can place their comments and notes in the source code. The one exception to this (and you will find that there are few exceptions in Python) is the very first line of code in this sample. The `#!/usr/bin/python` command lists the path to the Python program files so that when the script is run, the computer knows where Python resides. This line is optional in Windows, but is normally required when running on a UNIX environment, and so it is included here.

Following the path and the comments is a short `while` loop. This line says, "Do whatever follows once." Then the print instruction follows. Notice that the print line is offset, or tabbed inwards. White space in Python actually serves a purpose; it places the `print` command within the jurisdiction of the `while` loop. Also notice the semicolon at the end of that line. Semicolons, as you've learned, are used to end a statement.

This source code can be found in the \CHAPTER2 folder on the accompanying CD. If you run the program, you will see "Hello!" printed to the screen.

C's "Hello World"

For comparison, let's see what Hello from C would look like. There are many different ways to get C to print a string, but typically the effort looks like the following:

```
############
# HELLO_C_1.cpp
# This program displays the string "hello!".
# It first Includes the stdio.h library, then creates a main,
# then creates a short loop, and then prints the string.
############
#include <stdio.h>

main()
{
  for(;;)
```

```
        {
            printf ("Hello!\n");
        }
}
```

The comments are the same as in Python, but besides the comments you can see that the program is very different. First, C doesn't have a built-in `print` function, so you need to import a library that does. stdio.h is short for Standard Input and Output. The library is standard and comes with most C compilers, but it will add a significant amount to the compiled binary.

Second, every C program needs a `main` statement, a place for the main program piece to run in. This means that `main()` must be declared before you can proceed any further. Then come the squiggly brackets {}. C uses brackets to separate blocks and section of code. Whatever is in `main` must be bracketed by squiggly brackets.

Then comes the loop (in this case a `for` loop), which serves the same purpose here as Python's `while` loop; the syntax is, of course, different. Again, squiggly brackets are needed to bracket off what belongs within the `for` loop.

Finally, we get to `printf`, a command from the stdio.h library that prints input to the screen. Notice that the string must be between both the parentheses () and the quotation marks "". There is also the semicolon (;) that follows the end of a statement, something in common with Python. The `\n` is actually an escape sequence that creates a new line once the "Hello!" is printed.

The key thing to notice is that with C there are a few extra steps:

- A library that can work with strings must be imported.
- There must be a `main()`.
- Brackets {} must separate code blocks.

Also look at the syntax. Any missed colon, semicolon, parenthesis, bracket, slash, or pound sign will result in a program error. Python's code has fewer symbolic syntax needs because the designers wanted something that would be easy to write and read.

The Lua Environment

The idea behind Lua is that it is to be used as a lightweight configuration language for any program that needs one. It is written in clean C, which means the Lua source code is made up of the most common subset of ANSI C and C++. Since this section is about Lua and this book isn't a book on C, I won't spend a lot of time going over any C code. If you want a primer on the C language, I suggest picking up a book on C; there are hundreds to choose from.

Lua is implemented as its own library. It's purely an extension language, and so it has no "main" loop of its own. Lua normally functions embedded within a host client, like C code or a C program. It is the host program that invokes Lua code, reads and writes Lua variables, and so on. Lua can also be extended by C and C functions. We'll look more at

combining Lua and C (and Lua's C API) in the next chapter, and we'll examine extending Lua, Python, and Ruby in Chapter 12.

The use of C in this book is actually pretty infrequent. In fact, all the code samples in this chapter should run fine in the Lua interpreter alone. If you come across something in C that doesn't make sense, don't get nervous; just move on. Eventually, all code will succumb to your will and prowess.

Normally Lua is used within a host language, and usually the host language is C. Lua can also be used alone, usually for quick glue programs or text-processing utilities. These standalone projects tend to rely heavily on the basic libraries Lua provides. Finally, there are applications that use Lua as a library. These apps tend to have more program code in C than in Lua, and they create interfaces to the Lua language within C.

In this chapter, almost all of the examples are pure Lua and can be run with the Lua interpreter. Using Lua within a host language or library is covered more in the next chapter, where I get down to using Lua in a game-programming environment, and also in Chapter 12, where I'll discuss extending and embedding high-level languages.

Installing Lua

Lua is free software, and the license is included in the CD folder (under Lua) along with the necessary packages for building and installing Lua 5.0. This includes a generic tar.gz for building Lua from scratch on most platforms and an .rpm (Redhat Package Manager) for Linux Red Hat. You can build Lua from the source on any UNIX-flavor machine with the provided make files.

In order to build Lua from the source on a Windows machine, you need a development environment like Visual C++ 6.0 or Cygwin, but luckily for you, the precompiled win32 executables and binaries are included in the LuaWin32.zip file. Instead of your building Lua from scratch, the zip file will provide a lua.exe executable that starts up the Lua Interpreter.

NOTE

CAUTION

The preconfigured lua.exe and luac.exe binaries are statically linked, so when developing real projects you will want to place these within the bin folder of the full Lua source tree. The libraries included should also be placed in the Lua lib folder so that they can link with one another. See the documentation on installing Lua at http://www.lua.org

Lua 5.0 was released in April 2003. Some new features in 5.0 include:

- Coroutines (collaborative multi-threading)
- Full lexical scoping (replaces upvalues)
- Metatables (replaces tags and tag methods)
- Support for true / false Booleans
- Weak tables

- New API methods
- New error handling techniques

The original Lua language (Version 1.1) was first publicly released in 1994. Way back then, the language was free for academic use, but commercial licenses had to be negotiated. However, no commercial negotiations ever occurred, and in Feb 1995, with Version 2.1, the license opened up to commercial use. For the most recent version of Lua, check with the Lua home page at http://www.lua.org.

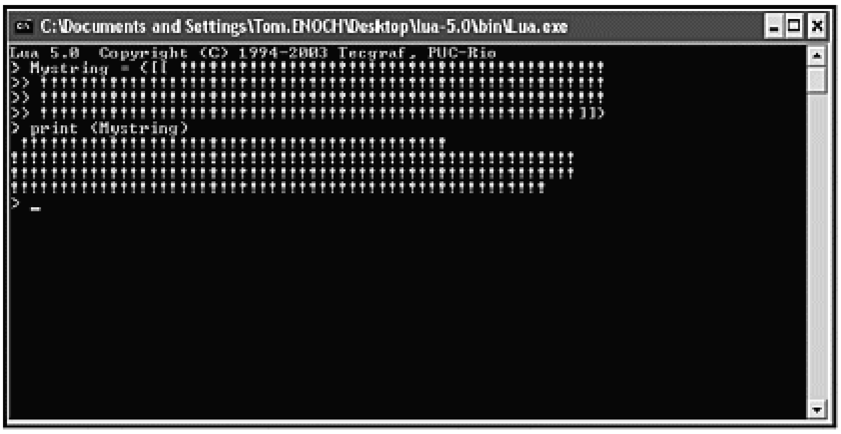
Or with Tecgraf at http://www.tecgraf.puc-rio.br/.

Lua was designed to run on anything out of the box. This versatility is a result of its plain vanilla C; you just need an ANSI C compiler to compile it. Lua should run not only on all standard Windows platforms, but also on UNIX, Linux, Solaris, SunOS, AIX, ULTRIX, and IRIX, not to mention NextStep, OS/2, Sony Playstation, Macs, BeOS, MS-DOS, OS-9, OSX, EPOC, and the PalmOS. Whew! Again, all you need is an ANSI C compiler to build Lua on the given platform.

The Lua Interpreter

The standalone interpreter (lua.exe on Windows machines) that comes with Lua is extremely useful, as it runs an interactive mode. When fired up, the interpreter displays the Lua version number and copyright notice at the top of the window, along with a greater than (>) symbol as a prompt (see Figure 2.14).

Figure 2.14. Opening the Lua standalone interpreter



In the interpreter, each command that you type executes immediately after you press the Enter key, and that line is considered to be a whole Lua chunk (more on Lua chunks in just a bit). The Lua interpreter is fairly smart, and if you need to enter multiple lines (for example, when creating a function), the Lua interpreter doesn't execute right away; instead, you will see two greater than symbols, indicating that the interpreter is waiting for you to end the function before executing (see Figure 2.15).

Figure 2.15. The multiple-line function in the Lua interpreter

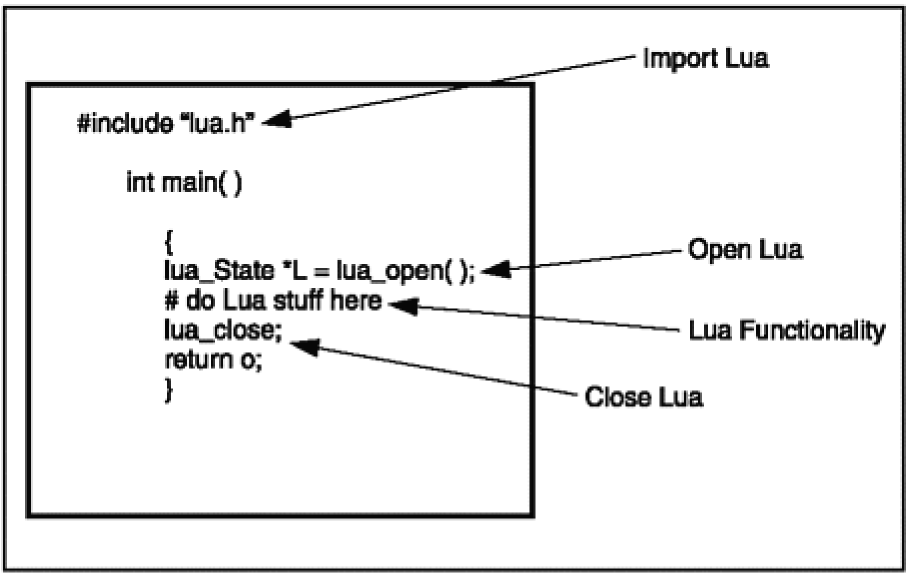


Most of the commands and samples in this chapter can be run in the interpreter, which is an excellent tool for getting a feel for Lua. I suggest you keep the interpreter open and try the sample code as you go along in the book.

Creating Lua Program Files

As I said, Lua is normally implemented via its host language. The host calls Lua with a `lua_open` command and then closes it with a `lua_close` command. A unit of Lua is stored in a file or string within the host program and is called a chunk. When the host executes a Lua chunk, the chunk is precompiled into bytecode for a virtual machine, and then the statements are executed in a sequential order. This Lua chunk does its thing, perhaps making changes to the global environment (that persist after the chunk ends), and then it ends (see Figure 2.16).

Figure 2.16. Lua being implemented via the C host language



NOTE

The term virtual machine (VM) was coined by Sun Microsystems to describe the runtime environment for their budding Java language. A VM acts as an interface between a compiled binary code and an operating system.

Lua has been designed as an extension language but it can be used as a stand-alone language as well. The Lua interpreter (named lua.exe) can be called via command line to execute Lua files (known by their .lua extension) and accepts a number of arguments, as shown in Table 2.7.

Table 2.7. Lua Interpreter Command-Line Arguments

Argument	Purpose
`-`	Executes `stdin` as a file
`-e stat`	Executes string `stat`
`-f file`	Requires file
`-i`	Enters interactive mode after running script
`-v`	Prints the version information
`--`	Stops handling options

If the Lua interpreter is given no arguments, it behaves as `if lua -`, or as `lua -v -i` when stdin is a terminal.

Chunks of Lua can be also precompiled into a binary form with Luac.exe, which is also included in the win32 executables. Luac.exe is a Lua bytecode compiler, an assembler that compiles the Lua source code into bytecode. This makes it completely unreadable to normal humans but also makes it run much faster. To use the bytcode assembler, you just call it as if you were compiling and then tell it what you want the new file to be (with a .lub extension) and what the sourcefile (.lua) is.

```
Luac.exe -o Myfile.lub Myfile.lua
```

The `-o` is one option to feed Luac, which means "output to file." For a full list of Luac options, see Table 2.8.

Table 2.8. Luac Options

Option	Purpose
`-l`	Produce a listing of the compiled bytecode for Lua's virtual machine
`-o "file"`	Output to file, instead of the default luac.out
`-p`	Load files but do not generate any output file
`-t`	Perform integrity tests of precompiled chunks

Table 2.8. Luac Options

Option	Purpose
`-v`	Print version information

NOTE

As Lua was written in ANSI C, you need to do special work when embedding Lua into a C++ application due to the "name mangling" that C++ performs. You must place `extern "C"` around the inclusion of Lua headers in a C++ application:

```
extern "C"{
# include "lua.h"
}
```

Lua has no if `def cplusplus` or `if def c` directives, because it is pure, clean ANSI C. This pureness makes the `extern` command necessary; without it, you will get link errors.

Lua's "Hello World"

Lua is quite different than the other two languages presented in this book. Lua is primarily an extension language, and Lua code is usually embedded within a host. You'll find Lua residing inside C, Python, and Ruby scripts, doing what it does best—acting as code within code. In-depth coverage of how to program with Lua is covered in Section 2 of this book, and what follows is just an example to whet one's appetite. With the understanding that a Lua "Hello World" program would normally exist within another language's construct, writing a "Hello World" program in Lua is even shorter and simpler than in Python or C:

```
--
-- HELLO_LUA_1.lua
-- This program displays the string "hello" .
-- It prints the string by using an internal print command.
--
print "hello world\n"
```

Notice that Lua's comments are different; they are marked by two dashes (--) instead of a pound (#) sign. Also notice that the print line itself is almost exactly like the C version Hello_C_1.cpp above, except in this case that you do not need to import a library for the `print` command, and a few of the symbols, namely the semicolons and parentheses, are left out.

The Ruby Environment

Ruby is most used to Posix type operating systems (such as UNIX, Linux, and FreeBSD) and is written in the C programming language. Although Ruby is comfortable

on UNIX, Linux, DOS, the various Windows flavors, Macintosh, and number of other platforms, it's most at home on the Posix environment where it was born.

Ruby on Windows needs a few additional tools in order to emulate its home environment. These tools include a Linux-like environment for Windows called cygwin, a collection of Windows header files and libraries called mingw, and the DJ Delorie software tools (djgpp). Precompiled versions of Ruby with these tools included can be found at the Ruby Central Website, which houses the Ruby "one-click" installer for Windows at http://www.rubycentral.com

The latest versions of this collection of tools can be found at their own respective Websites as well:

- **cygwin.** http://www.cygwin.com/.
- **mingw.** http://www.mingw.org/.
- **djgpp.** http://www.delorie.com/.

This Windows one-click installation is also included on the CD that accompanies this book, and can be found in the Ruby folder: \RUBY

Installing Ruby

The latest version of Ruby, 1.8.0 as of this writing, can be downloaded from the Ruby language organization Website at http://www.ruby-lang.org.

Developers can also take a peek at the source tree at that location. Ruby Version 1.8.0 is also on this book's CD in the \RUBY folder.

Windows users can simply use the one-click installer executable to install Ruby on their machines; just double-click on ruby180-10.exe to run the Ruby Setup Wizard. You may have to restart your computer afterwards.

Steps for installing Ruby on a Posix environment will vary, depending on the platform and also on any extension or static module linking that needs to be done. The following condensed steps will suffice for most folks, however:

1. Become a super user or user with privileges for installing new programs.

2. Run `autoconfto` generate configure.

3. Run `./configure` to generate config.h and the makefile.

4. Run `make`.

5. Run `make install`.

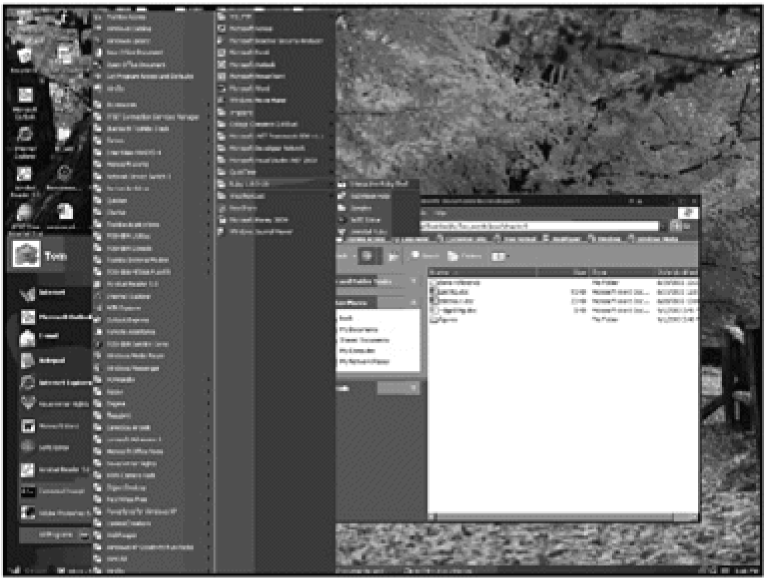
The Ruby Interpreter

Ruby can be used interactively with the interpreter, called irb, that comes bundled with it. For UNIX machines you need to add irb/ to the `$RUBYLIB` environment variable and

make a symbolic link to the irb.rb file in your path environment. Then you can type in irb to call the interactive Ruby shell.

On Windows, the irb is installed by default in the program file's directory, and the Ruby shell is accessible through the Start menu under Programs (see Figure 2.17). The code samples in this chapter can be run with the Interactive Ruby Shell.

Figure 2.17. Launching the Ruby interpreter from the Program menu



A program called eval, which is included in the samples/directory of the Ruby distribution, allows you to enter expressions and view their values.

Creating Ruby Program Files

Ruby program files invariably end with an .rb extension. They can be created in Notepad or vi or any other sort of text editor. To make things even easier, Ruby comes bundled with a nifty tool for scripting called the SciTE, which is a Scintilla-based text editor. SciTE has features for building and running many kinds of programs (see Figure 2.18), and it understands the syntax of a smattering of different computer languages, including Python, Lua, and Ruby.

Figure 2.18. The SciTE editor shows off its knowledge of Ruby syntax.



Executing Ruby

Ruby itself (that is, Ruby.exe) is meant to run on the command line, whether it's the UNIX shell or Windows command or DOS. The basic syntax for running Ruby is as follows:

```
Ruby options MyProgramScript arguments
```

Being a child of the command line, Ruby accepts a number of fun command-line options, or switches; these are outlined in Table 2.9.

Ruby comes with a Ruby Windows executable called rubyw.exe that will run on a Windows environment without launching a DOS or Windows command-line window, but the Windows platform will need to have .rb files associated with the executable for launching.

Ruby is primarily used as an interpreted language or as an extension. One extremely common use is to find Ruby on a server machine like a Web server where it is used as an interpreted language to run CGI or create Web forms and cookies. Ruby can also be embedded into HTML documents, another common use of the language.

Table 2.9. Ruby Command-Line Switches

Argument	Purpose
`-0digit`	Specifies the input record separator ($/) as an octal number
`-a`	Turns on auto-split mode
`-c`	Checks the syntax of the script and then exits without executing
`-Kc`	Specifies the KANJI (Japanese character) code-set

Table 2.9. Ruby Command-Line Switches

Argument	Purpose
`-d`	Turns on debug mode
`--debug`	Turns on debug mode
`-e`	Used to specify a script from the command line
`-F`	Used to specify the input field separator
`-h`	Prints a summary of all the command options
`--help`	Prints a summary of all the command options
`-I`	Specifies in-place-edit mode
`-l`	Enables automatic line-ending processing
`-n`	Used to run multiple iterations around the given script (looping)
`-p`	Same as `-n` but prints the value of variable `$_` at each end of the loop
`-r`	Causes Ruby to load a given file using `require`
`-s`	Enables some switch parsing for switches
`-S`	Forces Ruby to use the `PATH` environment variable to search for script
`-T`	Forces taint type checks to be turned on at the given level
`-v`	Enables verbose mode
`--verbose`	Enables verbose mode
`--version`	Prints the Ruby version
`-w`	Enables verbose mode without printing the version message at the beginning
`-x[directory]`	Tells Ruby that the script is embedded in a message and switches to a given directory (if provided) before executing a script
`-X`	Causes Ruby to switch to a given directory
`-y`	Turns on compiler debug mode
`--yydebug`	Turns on compiler debug mode

The most common use for games is to have Ruby associated with C as an extension. The Ruby interpreter is embeddable, and it is possible to embed the entire Ruby interpreter into C or other code. Just like Lua, Ruby has a full C API, which I'll cover in Chapter 10, and it is extendable not only with C but with other languages; I'll discuss doing that in Chapter 12.

Ruby's "Hello World"

A "Hello World" program in Ruby looks a lot like Python's:

```
#!/usr/bin/ruby
```

```
############
# HELLO_RUBY_1.ruby
# This program displays the string "hello" .
# It first shows the path to ruby, and then prints the string.
############
puts "Hello!"
```

Ruby's code is extremely streamlined in this example. A built-in `puts` command handles the printing without the need of any loops, spacing, brackets, or semi-colons. Very clean, this script simply tells the computer where Ruby is and then, in one line, tells it what to do.

Summary

I've covered quite a bit in this short chapter. Before you move on to more specifics with Python in the next chapter, or the other languages later on, you'll want to be sure you are familiar with Boolean logic and general program flow and structure with conditional and iterative control constructs. Important points from this chapter:

- Python, Lua, and Ruby organizations keep active lists of projects that are pretty extensive.
- Math and algebra are handled very similarly in each language.
- Boolean operators, Boolean comparisons, conditional control statements, and iterative control statements can all be used to control the flow of a program.
- Lists, strings, and a number of other commands all look and are handled in a similar way in each language.
- Implementing "Hello World" in a standard way in C takes more lines of code and more symbols than any of the other three languages.

Questions and Answers

1: Q: Why are there more projects in Python and Lua than Ruby?

A: A: Ruby is probably the most difficult of the three languages to find evidence of in the game industry. This is partly due to the language barrier (again, most modern Ruby development is in Japanese) and also because shops today tend to be using Ruby more for projects with the World Wide Web, XML integration, text processing, and general scripting. That doesn't mean Ruby isn't suited for game development; quite the contrary, as you will shortly see.

2: Q: I already know how to program "Hello World", when do I get to write graphics and games?

A: A: You'll start writing much more in-depth code in the very next chapter.

Exercises

1: Describe the difference between a conditional control statement and an iterative control statement.

2: Boolean logic uses only two values. Which two values are they?

3: Which `else/if` structure (`elseif`, `elsif`, and `elif`) goes with which language (Python, Lua, and Ruby)?

4: When printing a simple statement (like "Hello World"), one of the three languages normally uses a `puts` command instead of a `print` command. Which one is it?

Part TWO: Programming with Python

The next three chapters are all about Python. This part of the book starts with an overview of the Python language and its syntax, then moves in to examine commonly used libraries for writing games in Python, including Pygame and PyOpenGL. Finally, a few real-world Python game projects are examined.

Chapter 3. Getting Started with Python

Latet anguis in herba

——Virgil (70-19 BC), Roman poet, "Aeneid" (Translation: There's a snake hidden in the grass.)

Let's jump right into programming with Python. I'll start with an introduction to a few useful tools and then give you a speedy overview of the Python language.

Python Executables

You can execute .py files once Python is installed on your machine, but that doesn't make your Python game programs universally playable. You still need to convert your scripts into a bundled executable for whatever platform you want to run on. Luckily, there are a few resources for accomplishing just that.

Packaging Python Code

When modules are imported in Python by other modules, Python compiles the relevant code into byte-code, an intermediate, portable, closer-to-low-level binary language form. This byte-code is stored with the .pyc suffix, short for Python compiled, instead of the typical .py.

Python's .pyc files correspond roughly to DLLs (dynamically loaded libraries) used in C. Regular .py modules can be used dynamically, too, but the compiled Python code is tighter and Python interprets the code at runtime when the file is imported.

Precompiling scripts is one way to speed up Python programs that need to import many modules. You can minimize a program's startup time by making sure source code is kept in directories where Python will have access to writing .pyc files.

You can also ship Python programs as .pyc files rather than as .py scripts. Since .pyc files are binary, they cannot be run as scripts, but they can be sent to the Python interpreter; simply add the name of the .pyc file the next time you run Python, like this:

```
Python runme.pyc
```

In order to build a compiled Python file from the Python interpreter, import the `compile` function from py_compile and run the `compile` command, like this:

```
from py_compile import compile
compile("script_to_compile.py")
```

Freeze

Freeze is a system that takes Python script files and turns them into modules packaged into C files. Originally Freeze was used as one way to ship Python source, but it is now mostly defunct, although it will still be available in Version 2.3 for backwards-compatibility. The compiled script that Freeze generates allows a Python program to ship without the source code in plain view and without using .pyc files. The benefits to Freeze are that you can ship Python as two .c files and a makefile instead of as a .py, and you can make Python runable on platforms that do not have Python installed. The downside is that Freeze doesn't work well initially with Tkinter and other Windows GUIs.

ActiveState

ActiveState is a company that focuses on applied open source. It creates development packages for software developers and provides resources for Perl, Python, and PHP development. ActiveState currently has a Python distribution called ActivePython. It also supports creating Python RPM (Red Hat Package Managers) installers, Windows complete installers, and a Visual Studio .NET IDE plug-in for Python. These services (some are free, others not) are available at the ActiveState Python Website, at http://www.activestate.com/Solutions/Programmer/Python.plex.

py2exe

The py2exe extension is an open source utility that converts Python scripts into executable Windows programs. The software is copyrighted by Thomas Heller but is freely distributable, and you'll find a copy with the license on the accompanying CD under Python/py2exe.

The extension is still under development but has expanded recently to include the ability to turn Python scripts into Windows NT-like services; it has been used to create a number of popular Python applications, such as wxPython, Tkinter, and pygame (you'll get to know these applications a bit better in the next chapter).

py2exe is a Distutils (Python Distribution Utilities) extension, and relies on the work by Greg Ward to make Python programs distributable (see the Disutils Website at http://www.python.org/doc/current/dist/). The Distutils are necessary for py2exe to work and are also included on the Python folder in this book's CD.

Python Debuggers

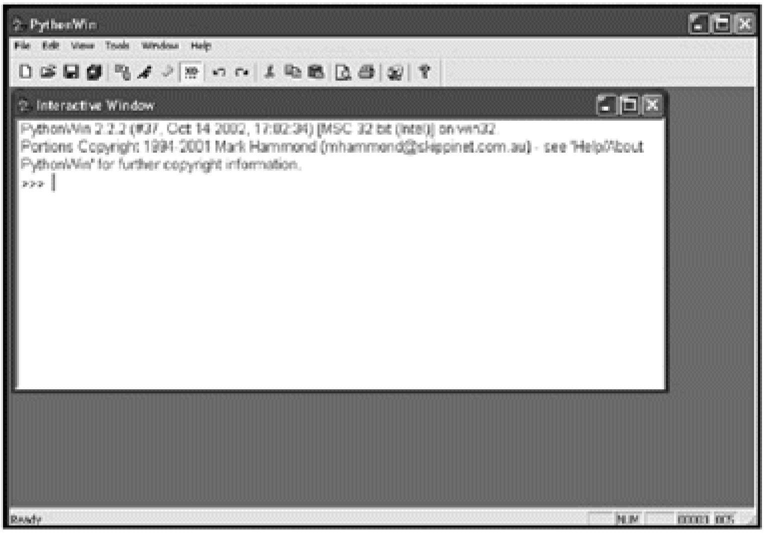
Python comes with a built-in pdb (Python Debugger) module that defines an interactive source code debugger for Python programs. The Python debugger supports a number of useful programming functions, such as breakpoint setting, stack frame inspection, source code listing, and so on, but unfortunately the debugger has been (historically) poorly documented and the windowing version module (wdb) is considered a bit primitive.

Since source-level debugging is such an important part of programming, a few improvements have been made to the existing Python debugger. Two popular free Python debuggers are commonly used. The first is PythonWin, the Python for Windows extension. Unfortunately it only runs on Windows. The second is the HAP (Humongous Addition to Python) debugger developed at Humongous.

PythonWin

PythonWin is a Python debugger and an IDE that runs on Windows. Versions for Python 2.2 and 2.3 are included on the accompanying CD under Python/debuggers. PythonWin is becoming the standard Windows debugger and is now included in some distributions of Python (for instance, in ActiveState's ActivePython). PythonWin has a GUI environment (see Figure 3.1) but can also be run via command line.

Figure 3.1. Opening shot of the PythonWin debugger



PythonWin is basically a wrapper for the MFC (Microsoft Foundation Class) libraries. PythonWin is copyrighted by Mark Hammond but is freely usable and distributable as long as the license (found in both the Licenses folder and the Python/Debuggers/PythonWin folder on the CD) accompanies the binary.

The Open Source HAP Debugger

HAP can be found among other open-source Sourceforge projects (http://hapdebugger.sourceforge.net/) and is released under the Gnu Lesser General Public Licenses (you can read the license in detail on the CD in the Licenses folder). The HAP debugger can be run remotely, which makes it an ideal tester for a computer game in a lab environment. The game can run on full screen on one machine while a second machine can debug it remotely.

The HAP debugger Version 3.0 is included on the CD, under the Python/debuggers/HAP folder. HAP was built with the idea that the debugger would move to console game development and development on the Macintosh, but currently it runs only on Windows 2000 and must be built with Visual C++. It provides a few features the standard Python debugger cannot, such as a full-screen mode and multi-threading. The debugger has two applications. The first is the editor and IDE, and the second is the remote debugging host. The first application runs whatever Python script is being debugged and then communicates to the IDE via a network socket.

NOTE

Consoles don't have keyboards, mice, or multiple monitors, so the default Python debugger isn't so great when you need to test console type games written in Python. This is one of the reasons Humongous developed the HAP debugger: remote debugging frees you from the platform and allows you to debug in a comfy computer environment.

NOTE

One Game Script's History

Before Humongous Entertainment used Python, they had an internal tool named SCUMM (Script Creation Utility for Maniac Mansion). Maniac Mansion was a project originally under LucasFilm Games, and SCUMM was the custom scripting language and game engine used to develop Maniac Mansion.

SCUMM was created by Aric Wilmunder and Ron Gilbert when they worked for LucasFilm. When Gilbert later founded Humongous and Cavedog Entertainment in the Pacific Northwest, he brought with him SCUMM, which the new companies used to create over 50 different games, including Humongous's original popular Freddi Fish, Putt Putt, and Pajama Sam children's titles.

SCUMM's limitations became too restricting after a decade or so of use, and at that time the company switched over to using C++ and Python for game development. The first game it scripted with Python was Backyard Hockey. The Game Logic, AI, menu, and actual executable of Backyard Hockey were all Python, which called in C++ modules for heavy graphics and sound when necessary.

Python Language Structure

Now that you can install and run Python in a variety of ways, it's time to get a real handle on the language itself. This section goes over Python's types, carries on from last chapter's section on math and loops, and also introduces a few new concepts.

Python Punctuation

As you have seen from the previous Hello World! examples, Python doesn't need a lot of punctuation. In particular, Python doesn't use the semicolon (;) to mark the end of line. Unlike C, Perl, or a number of other languages, the end of a line is actually marked with a newline, so the following is a complete command in Python:

```
print "hello"
```

Code blocks are indicated in Python by indentation following a statement ending in a colon, for example:

```
if name == this is true:
# run this block of code
else:
# run this block of code
```

Getting used to white space that actually means something is probably the most difficult hurdle to get over when switching to Python from another language.

NOTE

CAUTION

UNIX, Windows, and the Macintosh Operating System all have different conventions for how to terminate lines in text files. This is an unfortunate feature of multi-platform programming, and since Python uses terminated lines as syntax, your Python scripts written in text editors may not work on different platforms. The Macintosh version of Python recently fixed this problem; it now checks line endings when it opens a file and adjusts them on a per-file basis. It may be possible to find or write a filter that substitutes end-of-line characters for different platforms. Compiling scripts to byte-code before platform-hopping is another possible workaround.

Language Types

Python includes a handful of built-in data types (see Table 3.1); the most commonly used of these data types are numbers, strings, lists, dictionaries, and tuples. Numbers are fairly obvious, although there are several different number types, depending upon the complexity and length of the number that needs to be stored. Strings are simply rows of letters. Lists are groups that are usually comprised of numbers or letters. Dictionaries and tuples are advanced variable types that are similar to lists and comparable to arrays in other languages. These types all have built-in operations, and some have built-in modules or methods for handling them.

Table 3.1. Built-In Python Data Types

Name	Data Held
`complex`	Complex numbers (see Table 3.2)
`dict`	Dictionary
`file`	File
`float`	Floating point number (see Table 3.2)
`hexadecimal(0x)`	Hexadecimal number
`int`	Integer (see Table 3.2)
`list`	List
`long`	Long integer (see Table 3.2)
`object`	Base object
`octal(0)`	Octal number (see Table 3.2)
`str`	String
`tuple`	Tuple
`unicode`	Unicode string

Numbers

Python has several basic numeric types; they are listed in Table 3.2.

Table 3.2. Python Basic Numeric Types

Type	Example
`integer`	`1`
`long integer`	`1111111L`
`floating point`	`1.1`
`complex`	`1.1j,.1`
`octal`	`0111`
`hexadecimal`	`0x1101`

Integers are the most commonly used math construct and are comparable to C's long integer. Long integers are size-unlimited integers and are marked by an ending L. Floating points are integers that need a floating decimal point and are equivalent to C's double type. Octal numbers always start with a 0, and hexadecimal integers always begin with a 0x in Python.

Numbers can be assigned just like you would in a high school algebra math problem:

```
X = 5
```

The basic math operators (+, -, *, /, **, %, and so on), which were listed in Chapter 2, can be used in the standard mathematical sense.

```
# Make x equal to 2 times 6
x = (2*6)
# Make y equal to 2 to the power of 6
y = (2 ** 6)
Print y
```

Python always rounds down when working with integers, so you if you divide 1 by 20 you will always get 0 unless you use floating point values. To change over to floating point math, simply place the decimal in one of the equation's numbers somewhere, like so:

```
# This will equal 0
x = (1/20)
print x
# This will get you a floating point
y = (1.0/20)
print y
```

In addition to your basic math operators, comparison operators (>, <, !=, ==, =, >=, and <=) and logical operators (and, or, not) can be used with basic math in Python. These operators can also compare strings and lists.

NOTE

The truncation, or "rounding down," during integer division is one of the more common stumbling blocks for new users to Python.

Python comes with a built-in `math` module that performs most of the complex constant functions. The more common constants are listed in Table 3.3.

Table 3.3. Common Functions from `math`

Function/Constant	Description
`pi`	The mathematical constant approximately equal to 3.14
`e`	The base of the natural logarithm (ln) approximately equal to 2.7
`find`	Finds the lowest index where the second string (argument) appears in the first

Python also has a built-in `random` module just for dealing with random numbers. A few of the more common `random` functions are listed in Table 3.4.

Table 3.4. Common `random` Functions

Function	Description
`seed`	Seeds the random number generator; default seed is the current time
`random`	Returns the next random number as a floating-point number between 0 and 1.
`randint`	Returns a random number between two given integers
`uniform`	Returns a random number between two given floating-point numbers
`choice`	Randomly chooses an element from the specified list or tuple

Strings

You designate strings in Python by placing them within quotes (both single and double quotes are allowed):

```
Print "hello" 'hello'
```

Strings store, obviously, strings of characters. Occasionally you will want to print a special character, like a quote, and Python accepts the traditional backslash as an escape character. The following line:

```
Print "\"hello\""
```

prints the word `hello` in quotes. A few other uses of the escape sequence are illustrated in Table 3.5.

Table 3.5. Python Escape Sequences

Sequence	Function
`\n`	Prints a newline
`\t`	Horizontal tab
`\b`	Deletes the last character typed
`\a`	System beep
`\\`	Prints a backslash
`\r`	Prints a carriage return

Like with variables, you can manipulate strings with operators. For instance, you can concatenate strings with the + operator:

```
# This will print mykonos all together
print 'my'+'konos'
```

Anything you enter with `print` automatically has a newline, `\n`, appended to it. If you don't want a newline appended, then simply add a comma to the end of the line with your `print` statement (this only works in non-interactive mode):

```
# These three print statements will all print on one line
print "I just want to fly",
print "like a fly",
print "in the sky"
```

Lists

Lists were introduced in the Chapter 1. In Python, lists are simply groups that can be referenced in order by number. You set up a list within brackets [] initially. Integer-indexed arrays start at 0. The following code snippet creates a list with two entries, entry 0 being `"Ford"`, and entry 1 being `"Chrysler"`, and then prints entry 0:

```
cars = ["Ford", "Chrysler"]
print cars[0]
```

In Python, there are a number of intrinsic functions, or methods, that allow the user to perform operations on the object for which they are defined. Common list methods are listed in Table 3.6.

Table 3.6. Common List Methods in Python

Operation	What it does
`list = range()`	Creates a list
`list.append()`	Adds an element to the end of the list
`list.insert(index, element)`	Inserts an element at `index`
`list.sort()`	Sorts the list
`del list[:]`	Deletes a slice or section of a list
`list.reverse()`	Reverses the list
`list.count()`	Returns the number of elements in list
`list.extend(list2)`	Inserts `list2` at the end of list
`list.remove()`	Removes an element from the list

So, for instance, you can add to the list simply by using the `append` method:

```
cars.append ("Toyota")
print cars
```

Or you can slice up lists by using a colon. Say you want to print just the first through the second item from the `cars` list. Just do the following:

```
print cars[0:2]
```

Lists can contain any number of other variables, even strings and numbers, in the same list, but cannot contain tuples or nested lists. Once created, lists can be accessed by name, and any entry in a list can be accessed with its variable number. You can also reference the last item in a list by using `-1` as its reference number.

```
# This line prints the last entry in the cars list:
print cars[-1]
```

You can also use the basic operators explained in Chapter 2 to perform logic on lists. Say you need to print the `cars` list twice. Just do this:

```
print cars+cars
```

Lists can also be compared. In a case like this:

```
[1, 2, 3, 4] > [1, 2, 3, 5]
```

the first values of each list are compared. If they are equal, the next two values are compared. If those two are equal, the next values are compared. This continues until the value in one is not equal to the value in the other; if all of the items in each list are equal, then the lists are equal.

NOTE

CAUTION

Characters in a string act just like elements in a list, and can be manipulated in many of the same ways, but you cannot replace individual elements in a Python string like you can with a list.

If you need to iterate over a sequence of numbers, the built-in function `range()` is extremely useful. It generates lists containing arithmetic progressions, for instance:

```
# This snippet assigns the numbers 0 through 9 to list1 and then
prints the,
list1=range(10)
print list1
```

It is possible to let `range` start at another number, or to specify a different increment:

```
# The following line assigns the numbers 5-9 to list2
list2=range(5, 10)
```

```
print list2
# The following line creates a list that jumps by 5s from 0 through 50
and assigns it to
list3
list3=range(0, 50, 5)
print list3
# The following line does the same only in negative numbers
list4=range(-0, -50, -5)
print list4
```

Tuples

Python also has a structure called a tuple. Tuples are similar to lists and are treated similarly, except that they are designated by parentheses instead of brackets:

```
tuple1 = ( a, b, c)
```

You don't actually need parentheses to create a tuple, but it is considered thoughtful to include them:

```
tuple1 = a, b, c
```

You can create an empty tuple by not including anything in parentheses:

```
tuple1 = ()
```

There is also a version of the tuple, called a singleton, that only has one value:

```
Singleton1 = a,
```

While lists normally hold sequences of similar data, tuples (by convention) are normally used to holds sequences of information that aren't necessarily similar. For example, while a list may be used to hold a series of numbers, a tuple would hold all of the data on a particular student—name, address, phone number, student ID, and so on—all in one sequence.

So what makes tuples so special and different? Well, for one thing, tuples can be nested in one another:

```
tuple1=(1,2,3)
tuple2=(4, 5, 6)
tuple3 = tuple1, tuple2
print tuple3
```

When you enter the last line and print out `tuple3`, the output is:

```
((1, 2, 3), (4, 5, 6)).
```

NOTE

TIP

For convenience, there is `tuple()` function that converts any old list into a tuple. You can also perform the opposite operation, using the `list()` function to convert a tuple to a list.

You can see how Python continues to bracket and organize the tuples together. Nesting tuples together in this way, also called packing, can provide a substitute for things like two-dimensional arrays in C.

There is one more interesting feature, called multiple assignments, in tuples.

```
X, Y = 0, 1
```

Python assigns X and Y different values, but on the same line of code. Multiple assignments can be very useful and quite a timesaver.

Dictionaries

Python has a third structure that is also similar to a list; these are called dictionaries and are indexed by assigned keys instead of automatic numeric list. Often called associative arrays or hashes in other languages, dictionaries are created in Python in much the same way as lists, except that they are used to create indexes that can be referenced by corresponding keys. An example of this might be a phone directory, where each telephone number (value) can be referenced by a person's name (key).

Dictionaries are designated with curly braces instead of brackets. The keys used to index the items within a dictionary are usually tuples, so you will see them put together often. You can create an empty directory in the same way you create empty tuples, except that you replace the parentheses with curly braces, like so:

```
dictionary1 = {}
```

You assign keys and values into a dictionary using colons and comas, like so:

```
key : value, key : value, key : value
```

So for instance, in the phone number directory example:

```
directory = {"Joe" : 5551212, "Leslie" : 5552316, "Brenda" : 5559899}
```

Then you can access specific indexes by placing the key into brackets. If I wanted to reference Brenda's phone number later on, the following snippet would do the job and give me 5559899:

```
directory [Brenda]
```

If I had mistyped the number, I could change it to new value like this:

```
directory[Brenda] = 5558872
```

Dictionaries have a number of standard methods associated with them; these are listed in Table 3.7.

Table 3.7. Common Dictionary Methods in Python

Operation	**What it does**
`clear()`	Deletes all items in a dictionary
`get()`	Returns key value
`has_key()`	Returns 1 if key is in dictionary, else 0
`keys()`	Returns a list of keys from dictionary
`update(dictionary2)`	Overrides the dictionary with values from dictionary 2, adds any new keys
`values()`	Returns a list of values

Identifiers

Identifiers are used in Python to name variables, methods, functions, or modules. Identifiers must start with a non-numeric character, and they are case sensitive, but they can contain letters, numbers, and underscores (_).

There are also a handful of words Python reserves for other commands. These are listed below:

```
and

elif

else

except

exec

finally
```

```
for

from

global

if

import

in

is

lambda

not

orassert

passbreak

printclass

raisecontinue

returndef

trydel

while
```

As a convention (but not necessarily a rule), identifiers that begin with two underscores (__) have special meanings or are used as built-in symbols. For instance, the `__init__` identifier is designated for startup commands.

Python's variables are loosely typed, and you can assign any type of data to a single variable. So, you can assign the variable `x` a numeric value, and then turn around later in the same program and assign it a string:

```
X=111
Print x
X="Mythmaker"
Print x
```

NOTE

Not realizing that Python's variable names are case-sensitive seems to be one of the most common mistakes new users to the language suffer from.

Control Structures

The very common `if`, `elif`, and `else` statements showed up in Chapter 2. These are used in Python to control program flow and make decisions:

```
if x ==1:
        print "odd"
elif x == 2:
        print "even"
else:
print "Unknown"
```

`if` can also be used with Boolean expressions and comparison operators to control which blocks of code execute. Unlike with most other languages, you'll see that parentheses aren't commonly used to separate blocks in Python, but colons, tabs, and newlines are.

```
if 1 >2 :
        print "One is greater than two"
else :
        print "One is not greater than two"
```

Loops

You saw how Python's `for` loop is used in Chapter 2. `for` is fairly versatile, and works with lists, tuples, and dictionaries.

```
for x in cars:
print x
```

The following example uses `for` to loop through the numbers 0–9 and then print them:

```
for x in range(0, 10) :
        print x
```

This same example can be rewritten with a `while` loop:

```
x = 0
while x <= 10 :
        print str(x)
        x += 1
```

The `else` clause will not fire if the loop is exited via a `break` statement.

A number of convenient shortcuts exist for use with Python `for` loops; you'll get used to using them after a while. For instance, Python will run through each item in a string or list and assign it to a variable with very little necessary syntax:

```
for X in "Hello":
# In two lines you can print out each item of a string
print X
```

You can use a few borrowed C statements in `for` and `while` loops in order to control iterations, including the `break` statement, which breaks out of the current `for` or `while` loop, and the `continue` statement, which jumps to the next iteration of a loop. You can also add to the loop an `else` clause that will execute after the loop is finished (in the case of `for` loops) or when the `while` condition becomes false (in the case of `while` loops).

```
x = 0
while x <= 10 :
        if x == 22:
                # this breaks out of this while loop
                break
        print str(x)
        if x <=11:
                # this jumps to the next loop Iteration
                continue
        x += 1
        else:
                # This happens when x <=10 becomes false
                break
```

NOTE

CAUTION

It's a common mistake, when first playing with loops, to create a never-ending loop that locks out any program control. For instance, the following code will never encounter a condition to exit and will therefore execute forever:

```
while 1 == 1:
     print "Endless loop."
```

Modules

Python is based on modules. What this means is that when a Python source file needs a function that is in another source file, it can simply import the function. This leads to a style of development wherein useful functions are gathered together and grouped in files (called modules) and then imported and used as needed. For instance, let's say the source file MyFile.py has a useful function called `Useful1`. If you want to use the `Useful1` function in another script, you just use an `import` command and then call the function, like so:

```
import MyFile
MyFile.Useful1()
```

For instance, create a file called TempModule.py with the following four lines:

```
def one(a):
        print "Hello"
def two(c):
        print "World"
```

This file defines two functions: the first function prints "Hello" and the second one prints "World". To use the two functions, import the module into another program by using the `import` command, and then simply call them, like so:

```
import TempModule.py
TempModule.one(1)
TempModule.two(1)
```

The `(1)` is included here because each function must take in one argument.

You can also use `dir()` to print out the functions of an imported module. These will include whatever has been added and also a few built-in ones (namely `__doc__`, `__file__`, `__name__`, and `__built-ins__`).

Module-based programming becomes particularly useful in game programming. Let's say you like the `Useful1` function, but it really hinders performance when it runs in a game because it makes a lot of intense graphical calls or does a lot of complex math. You can fix `Useful1` by simply rewriting the necessary functions and typing in MyFile.py as C++ code (or another language like assembly) and then registering the functions with the same module name. The original Python script doesn't even have to change; it just now calls the new, updated, faster C++ code. Modules make it possible to prototype the entire game in Python first and then recode bits and pieces in other, more specialized programming languages.

Python has a large selection of modules built into the default distribution, and a few of the commonly used ones are listed in Table 3.8.

Table 3.8. Commonly Used Built-In Modules

Module	**Description**
`sys`	Basic system and program functions
`argv`	List of commands to be passed to the interpreter
`stdout`, `stdin`, and `stderr`	Basic standard output, standard input, and standard error
`exit`	Exits the program gracefully
`path`	The paths Python looks at to find modules to import

Libraries

Python ships with a number of great, well-documented libraries. Some of these libraries are providers of Python's much-celebrated flexibility. The library list is constantly growing, so you may want to check out the Python library reference below before embarking on any major projects:

http://www.python.org/doc/current/lib/lib.html

Creating a Simple User Interface in Python

There are two simple functions in Python for getting keyboard input from the user: `raw_input` and `input`. The `raw_input` function is the easier to use of the two. It takes one argument and waits at a normal keyboard prompt for a user to type something. Whatever is typed is then returned as a string:

```
X = raw_input( "Enter your name: " )
print X
```

`input` works just like `raw_input` except that it is preferable to use with numbers because Python interprets the variables as whatever is typed in, instead of converting any numbers into strings.

```
X=input("Enter a Number: ")
print X
```

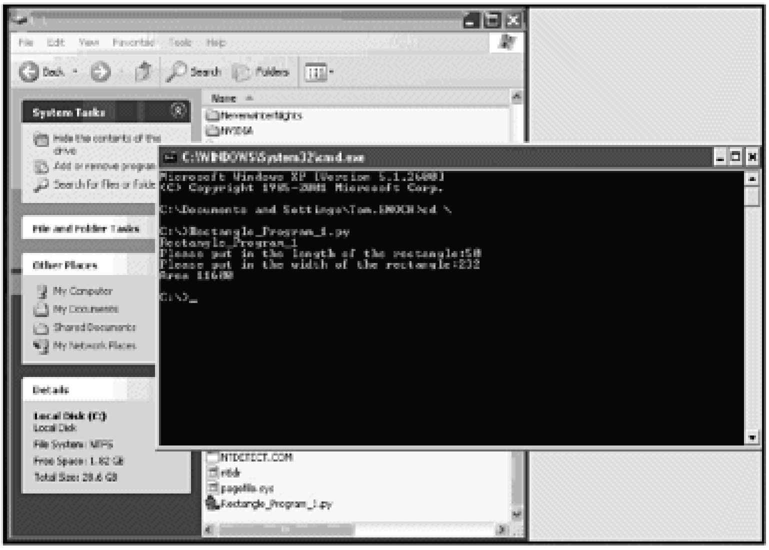
`input` will think that everything that has been entered is some sort of number, so if you enter in a string by using `input`, Python will conclude that the string represents a number.

Let's try `input` with something a bit more complex—a bit of code that calculates the area of a rectangle. To do so, you simply need two input lines and then a `print` statement that displays the results:

```
# This program calculates the area of a rectangle
print "Rectangle_Program_1"
length = input("Please put in the length of the rectangle:")
width = input("Please put in the width of the rectangle:")
print "Area",length*width
```

You can find this program, called Rectangle_Program_1, in the Chapter 3 folder on the CD. When you run it, it spits out output similar to you can see in Figure 3.2.

Figure 3.2. Python calculates the area of a rectangle based on user input



Here is a `while` loop in action with `input`:

```
# This program adds until the user quits
a = 1
sum = 0
print "Enter Numbers to add:"
print "Enter Q to quit."
while a != 0:
        print 'Current Sum:',sum
        a = input('Number? ')
        sum = sum + a
print 'Total Sum =',sum
```

This loop takes in numbers from a user and keeps adding them until the user quits with a Q entry. You'll see there is nothing new here; you're just mixing two functions from the chapter together. You can also find this code sample on the CD as Addition_1.py.

A Simple GUI with Tkinter

The GUI API approved by Python is a nifty toolkit already familiar to folks in UNIX land, TCL (short for Tool Command Language). Tkinker is the library behind common Python window interfaces and its version of TCL. Folks familiar with TCL\Tk in UNIX will find that Tkinter is very familiar. Although the library gets more extensive coverage in Chapter 4, I'll give you a small taste of a GUI that will run on a standard Windows environment to whet your appetite.

NOTE

Other GUI packages besides Tkinter are available for use with Python. For instance, the C STDWIN package is somewhat popular. However, Tkinker is the standard and comes shipped and installed with each Python package, so it's generally the first graphical tool folks discovering Python learn to use.

GUIs in Python are built from GUI components. In Windows, these components are called Windows Gadgets, or widgets for short. These widgets are listed in Table 3.9.

Table 3.9. Tkinter Widget Components

Component	Function
`Button`	Creates a button that triggers an event when clicked
`Canvas`	Displays text or images
`Checkbutton`	Creates a Boolean check button
`Entry`	Creates a line that accepts keyboard input
`Frame`	Creates the outlying window's edge
`Label`	Displays text as labels for components
`Listbox`	Creates a list of options
`Menu`	Creates a multiple-selection display
`Menubutton`	Creates a pop-up or pull-down style menu
`Radiobutton`	Creates a single option button
`Scale`	Creates a slider that can choose from a range
`Scrollbar`	Creates a scrollbar for other components
`Text`	Creates a multiple line box that accepts user input

When using Tkinter, you start by importing the library and then creating a frame that houses all of the other components:

```
From Tkinter import*
window = Frame()
```

If you run this via a script, or from Python's interactive mode, you will see an empty Tkinter window box appear, as shown in Figure 3.3.

Figure 3.3. Tkinter produces an empty frame widget



Let's add a simple label and a quit button to the widget. You will not be able to run this code in an interactive environment; you will have to actually create a file with a .py extension and run it via command line, DOS prompt, or by double-clicking it. For reference, the completed script can be found in the Chapter 3 folder on the CD.

To add a label, you need to first add the `pack` method. The `pack` method is used to determine the size and influence of a given component:

```
window.pack()
```

After referencing the `pack` method, you can add the `Label` method and specify the text (`'Hello'`) and placement (`TOP`) inside parentheses:

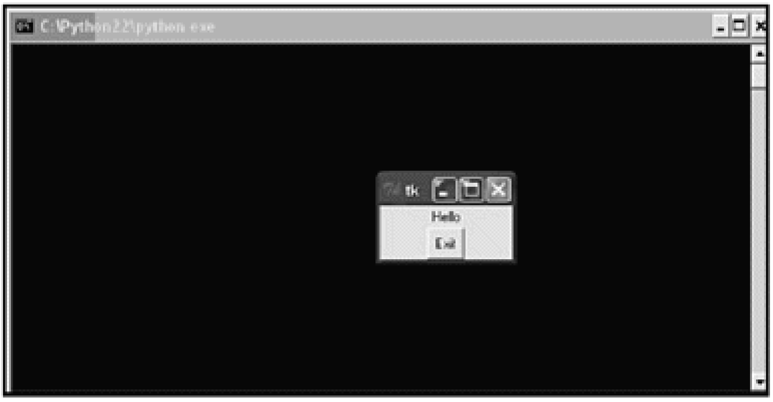
```
Label(window, text='Hello').pack(side=TOP)
```

Finally, you add a button using the `Button` method, specifying the text (`'Exit'`), the command the button will execute (`.quit`), and then you tell the `pack` method where to place the button (`BOTTOM`):

```
Button(window, text='Exit', command=window.quit).pack(side=BOTTOM)
```

One last step is to use the `mainloop` method to start the event loop. The full code snip follows and produces something similar to that in Figure 3.4:

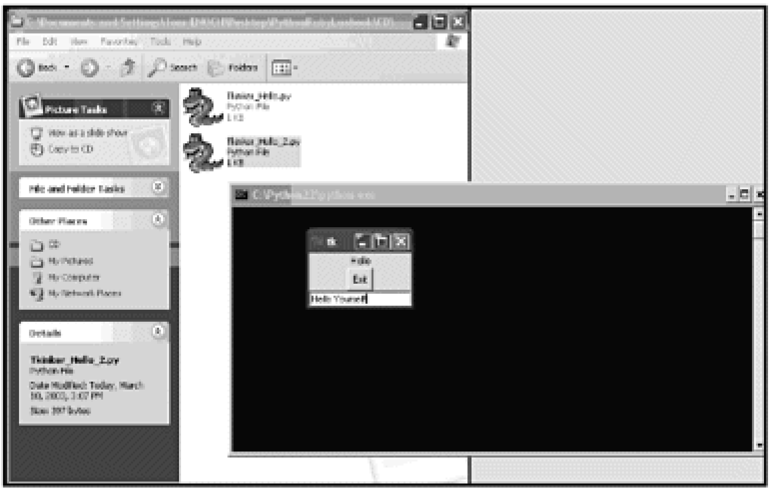
Figure 3.4. Tkinter says "Hello" with a slightly more complex widget




```
from Tkinter import *
window = Frame()
window.pack()
Label(window, text='Hello').pack(side=TOP)
Button(window, text='Exit', command=window.quit).pack(side=BOTTOM)
window.mainloop()
```

To create a simple user interface utilizing Tkinter, you will need to take advantage of Tkinter's `Entry` component. `Entry` works just like `raw_input` and will take what a user types in and return it after the Enter key is pressed. A simple Entry box can, by adding a line to `Tkinter_Hello`, specify a name for the widget and tell `pack` how to display it. Adding this line above the `mainloop()` command in `Tkinter_Hello` will give you an entry in the window you created to type into, as shown in Figure 3.5 (this code sample is also on the CD as `Tkinter_Hello_2.py`).

Figure 3.5. Now the Tkinter widget also has an entry box for typing into



```
Entry(name = "text1").pack(expand = YES, fill = BOTH)
```

Memory, Performance, and Speed

In Python, everything is an object, and all objects are allocated in dynamic memory (also called the heap). Because all objects are reference counted, you don't have to worry about freeing memory yourself; this is one of the great benefits of a high-level language. But if you're writing a game, especially a game that has to operate on a PDA or console, you may have to worry about memory allocation and memory fragments.

The Garbage Collector

The first issue is garbage collection. Traditionally, a game's biggest problem is with memory locks that get used up by the game process but not released back to the computer—that is, memory leaks. When a variable goes out of scope or is deleted, it needs to move toward being freed from memory. Problems can arise, however, if a variable is referencing a number of objects—these extraneous objects may keep the variable from being deleted. The worst-case scenario is when object A is referencing object B and vice versa, in which case neither object can be deleted. Since Python automatically reference-counts each object, this isn't a giant problem. Python's garbage collector will sweep through all objects eventually and clean them up. However, Python's collector will not automatically pick up references to unwanted objects or unclosed files. Failure to delete references to unused objects and leaving unused files open could cause memory leaks to occur. As a rule, all resources in a program should be released as soon as they are no longer needed.

Another potential problem with automatic garbage collection is that as a programmer, you have zero control over when the collector runs. If the collector decides to run while an important level-loading movies sequence is occurring, or during an unusually intense graphic sequence, your game could lose flow or its frame rate could be lowered. One solution to this is to temporarily disable Python's garbage collector while the game is running and then explicitly call it when you want it.

Access Python's garbage collector with the `gc` (short for Garbage Collection) module. Python's garbage collector is capable of reporting on how many unreachable objects are still allocated memory (this feature is called the Cycle Detector) or how many objects it is currently tracking. These methods (and others) are listed in Table 3.10.

Table 3.10. Commonly Used `gc` Functions

Function	Purpose
`collect()`	Does a full memory collection
`disable()`	Turns automatic garbage collection off
`get_debug()`	Gets debug flags
`get_objects`	Returns a list of the objects the collector is tracking
`get_referrers`	Returns a list of objects that refer to other objects
`get_threshold`	Returns current collection threshold
`garbage`	Where Python places cyclic garbage with finalizers

Table 3.10. Commonly Used `gc` Functions

Function	Purpose
`enable()`	Turns automatic garbage collection on
`isenabled()`	Returns true if automatic garbage collection is on
`set_debug()`	Sets debug flags
`set_threshold`	Sets the collection threshold

Several constants are also provided for use with `set_debug()`, as shown in Table 3.11.

Table 3.11. `set_debug` Constants

Constant	Use
`DEBUG_STATS`	Print statistics during collection
`DEBUG_COLLECTABLE`	Print information on any collectable objects found
`DEBUG_UNCOLLECTABLE`	Print information of any uncollectable objects found
`DEBUG_INSTANCES`	Print information about instance objects found
`DEBUG_OBJECTS`	Print information about objects other than instance objects found
`DEBUG_SAVEALL`	When this flag is set, all unreachable objects found will be appended to garbage rather than being freed
`DEBUG_LEAK`	Print information about a leaking program

You can use the `del` command to forcibly remove an object from memory. However, `del` is a finalizer; if you use it on an object, the garbage collector can no longer play with that object, and it loses control. So be sure you know what you are doing.

NOTE

CAUTION

Python's cyclic garbage collector is new as of Python 2.0, and the gc API was added in Version 2.2. Earlier versions of Python will not be as pliable where garbage collection is concerned.

NOTE

TIP

The `stack_dealloc` function is what Python uses as a destructor to clean up memory blocks after they have been designated. This frees up the memory in `PyMem_DEL`, the

space that holds objects that are decrementing toward deletion. However, if you aren't familiar with C style `malloc` type commands or memory management on a base level, you should probably hold off on forcibly clearing memory.

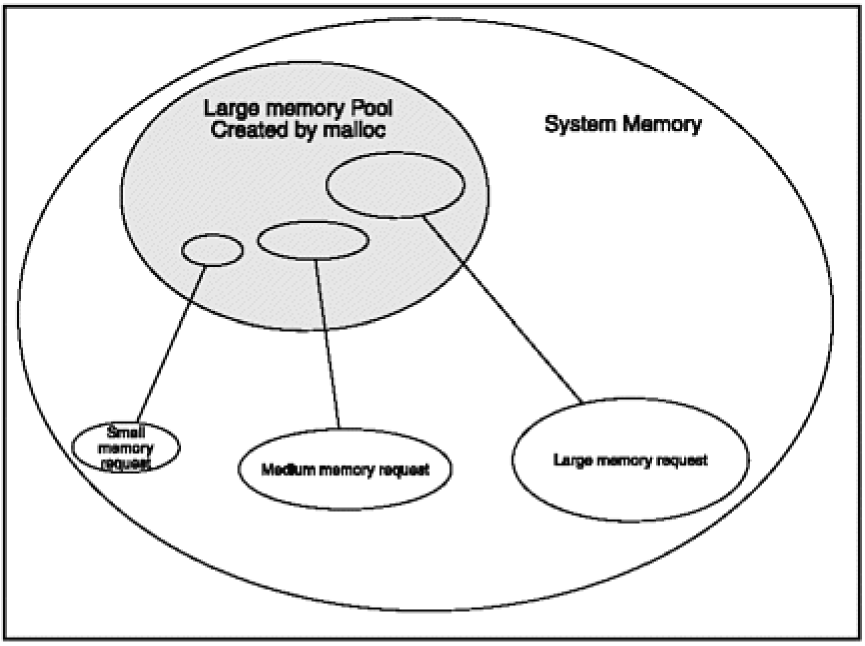
Pool Allocators

Another concern, particularly with consoles, is keeping Python memory allocation contained. Using memory or the garbage collector carelessly can cause Python to swoop in and eat up all a machine's available virtual memory. The trick is to isolate Python into its own memory arena.

Luckily, a few new and upcoming features exist in Python that help out with this issue. Pymalloc, an experimental feature added by Vladimir Marangozov in Version 2.1, is one of these. Pymalloc is a specialized object allocator that actually utilizes C's `malloc()` (short for memory allocation) function to get large pools of memory and then fill smaller requests for memory from these pools. Since Pymalloc is optional in Version 2.1 and 2.2, you need to include an option to the configure script (in the form of `--with-pymalloc`) in order to use it. Python Version 2.3 or higher enables it by default.

Pymalloc works by dividing memory requests into size classes (see Figure 3.6). These classes range from eight to 256 bytes and are spaced eight bytes apart. Memory requests lie within 4k pools that hold requests. Pymalloc allocates and deallocates requests for memory from these classes within pools. When deallocating Pymalloc memory classes, the classes can be completely freed (using `free()`) or released back into their respective pools. When the pools are empty, they are also released back into the memory at large.

Figure 3.6. Pymalloc doles out memory requests



NOTE

CAUTION

Pymalloc is meant to be transparent, but it may expose so-far-unknown bugs when used with C extensions. There have already been documented problems using Pymalloc with Python's C API. Use with caution.

Besides Pymalloc, in Version 2.3 Python has deprecated the previous API for dealing with memory and has new functions, some under PyMem, for allocating memory by bytes or type, and some under PyObject for allocating memory specifically for objects.

Performance and Speed

If you write Python code to do complex numerical work and then compare the results to those done with C++, you will be disappointed. The plain truth is that Python is a slower language. In Python, every variable reference is a hash table lookup, and so is every function call. This cannot compete with C++, in which the locations of variables and functions are decided at compile time.

However, this does not mean that Python is not suitable for game programming; it just means that you have to use it appropriately. For instance, if you are doing string manipulations or working with maps, Python may actually be faster than C++. The Python string manipulation functions are actually written and optimized in C, and the reference-counted object model for Python avoids some of the string copying that can occur with the C++ string class.

And, as I mentioned before, even if you don't think you should write your polygon collision detection code in Python, you may want to write your AI code and game loop in Python and prototype the collision detection. Then, after benchmarking, you can write the collision detection in C++ and expose it to Python. This will make coding much faster for you.

The Python `profile` module can be used to profile sets of functions. If you had a function called `MyFunction` stored in `MyModule`, the function can be imported into new script or the Python interpreter and then profiled by running:

```
import MyModule
profile.run('MyFunction()')
```

Python's `profile` module prints a table of all the function calls and each function call's execution time. Python also possesses a useful `trace` module that can be used to trace the execution of Python scripts.

You'll find that most folks will argue against using Python in games for speed-related issues more than any other. Here are a few performance tips to wrap up the chapter and to keep in mind for dealing with speed issues:

- Python has a number of debugging tools to use for benchmarking. If you get used to using them, you can easily get a feel for where things are slow in a given program.

- Be careful when using loops, since multiple iterations can easily become memory hogs. Systems calls should be moved outside of loops whenever possible (actually, systems calls should be avoided if at all possible). Try not to instantiate any objects inside of loops; doing so can cause many copies in memory and lots of work for the garbage collector.
- Use references instead of actual values when calling values, unless the values are very small.
- Avoid passing long argument lists to functions and subroutines. Keep them short and simple.
- Avoid reading or writing files line by line. Read them into a buffer instead.
- Check out all the fun libraries before building a function, and in particular, pay close attention to what Python has built in. Your newly written function is probably slower than the version the community has been using for a few years.
- Pay close attention to Chapter 12 in this book and learn how to extend Python in C.
- Use the `-o` switch when compiling to Python to byte-code (o is short for the compiler optimizing mode)
- Use aliases for imported functions instead of using the full name. Again, be especially careful when you do things like use full names inside of a loop.
- C++ programmers sometimes joke about optimizing their code by making variable names shorter. In Python this may actually work, since Python looks up variables by name at runtime.
- Avoid `while` loops with a loop counter. Instead use `range()` or `xrange()`. The Python `range()` operator is fast because it actually constructs a sequence object over which to iterate.
- Avoid heavy use of module-scoped variables. Locally scoped variables are usually faster.

Finally, keep in mind that optimizing code can take a lot of time and effort and isn't always worth it. Also, optimizing may cause other, bigger problems, such as making code harder to maintain, harder to extend, or buggier. Only if a script is running hundreds of times a day, or if the code relies on speed as a requirement, is shaving a few seconds off of it worth the development time.

Summary

Before you move on to the next chapter, make sure that Python installed correctly and that you can run the interactive environment. You should feel pretty comfy with `for` loops, `while` loops, and `if else` statements. You should also have had a chance to poke around with Tkinter a bit, and you should know what Python's garbage collector does.

Important points from this chapter:

- Tabs, colons, and newlines are the basis for Python punctuation.
- Python's most common data types are numbers, strings, lists, tuples, and dictionaries.
- Modules can be used to pass functions from file to file.
- Garbage collection and other aspects can be managed in Python if necessary.

Questions and Answers

1: Q: I've heard that the creator of Python and other writers have written tutorials on speeding up Python execution. Why aren't they mentioned here?

A: A: This chapter is a quick, whirlwind introduction to the Python language and is not meant to be an all-inclusive guide. Both Guido van Rossum and Andrew Dalke have written a few great online articles on benchmarking, Python performance, and other topics. The Python Essays Web page is a good place to start looking into the topic; it's at http://www.python.org/doc/essays/.

2: Q: Is there more to Python graphics than just Tkinter?

A: A: Absolutely. These are covered in the next chapter, along with a closer look at the usefulness of Tkinter.

3: Q: What about music? Does Python have any functionality for sound built in?

A: A: Python does have libraries that work with music and sound effects. I cover the Musickit library in Chapter 4.

Exercises

1: Lists use parentheses and tuples use brackets for assignments. What do dictionaries use?

2: List two escape sequences.

3: Name any one list method and what it does.

4: Write a simple example of a `for` loop.

5: Define "widget."

6: List one possible action that could slow down a program when used within an iteration, or loop.

7: Write a program that takes as input two strings and two integers and then displays them.

Chapter 4. Getting Specific with Python Games

…corporate methods do not have the conceptual framework to deal with an anarchist collective run by intelligent and arrogant comedians who have proved that their method works.

——Robert Hewison on the Monty Python group.

Now that you've completed Chapter 3's quick tutorial, it's time to jump into a few specific multimedia Python libraries and script an actual game or two. This chapter gets started with Python's Pygame library and moves specifically into graphics, networking, and sound for game programming.

The Pygame Library

Pygame is a Python wrapper for the Simple DirectMedia Layer (SDL). Pygame focuses on bringing the world of graphics and game programming to programmers in an easy and efficient way.

Typically, Pygame projects are small, simple, two-dimensional or strategy games. In Chapter 5, I'll give you a close look at a few existing Pygame-based game engines, including Pyzzle, a Myst-like engine; PyPlace, a two-dimensional isometric engine; and AutoManga, a cell-based anime-style animation engine.

Installing Pygame

This book's CD comes with a copy of Pygame in the \PYTHON\PYGAME folder. The most recent versions can be found online at http://www.pygame.org/download.shtml.

The Windows binary installer on the CD has versions for Python 2.3 and 2.2, there is a Mac .sit for older Mac versions and a version for the Mac OSX, and the RPM binary has been included for the Red Hat operating system. Pygame actually comes with the most recent and standard UNIX distributions, and can be automatically built and installed by the ports manager.

On Windows, the binary installer will automatically install Pygame and all the necessary dependencies. A large Windows documentation package, along with sample games and sample code, is available at the Pygame homepage at http://www.pygame.org.

Pygame also requires an additional package, called the Numeric Python package, in order to use a few of its sleeker and quicker array tools. This package can be found in the Python section of the accompanying CD. At this time, the creators of Numeric Python are working

NOTE

SDL

SDL is considered an alternative to Direct X especially on Linux machines. As a multimedia and graphics library, SDL provides low-level access to a computer's video, sound, keyboard, mouse, and joystick.

SDL is similar in structure to a very rudimentary version of Microsoft's Direct X API, the big difference being that SDL is open source, supports multiple operating systems (including Linux, Mac, Solaris, FreeBSD, and Windows), and has an API binding to other languages, including Python.

SDL is written in C and available under the GNU Lesser General Public License. Sam Lantinga, who worked for both Loki Software and Blizzard entertainment, is the genius behind SDL. He got his start with game programming in college by porting a Macintosh game called Maelstrom to the Linux platform.

Sam was working on a Windows port of a Macintosh emulator program called Executor and figured that the code he was building to extract the emulator's graphics, sound, and controller interface could be used on other platforms. Late in 1997 he went public with SDL as an open-source project, and since then SDL has been a contender.

on an even faster version called Numeric. If you need to install the Numeric package, use the .exe for Windows or the tar.gx for Posix environments. Numeric is distributed under an OSI license just like Python itself, and the latest development can be found at http://sourceforge.net/projects/numpy.

The Mac OS X tar includes Python 2.2, Pygame1.3 (hacked for Macs), PyOpenGL, and Numeric. There are still some bugs and issues with SLD compatibility on pre-OS X and post-OS X, and a simple installer for the Mac that should fix most of these issues is planned for when Python 2.3 is released.

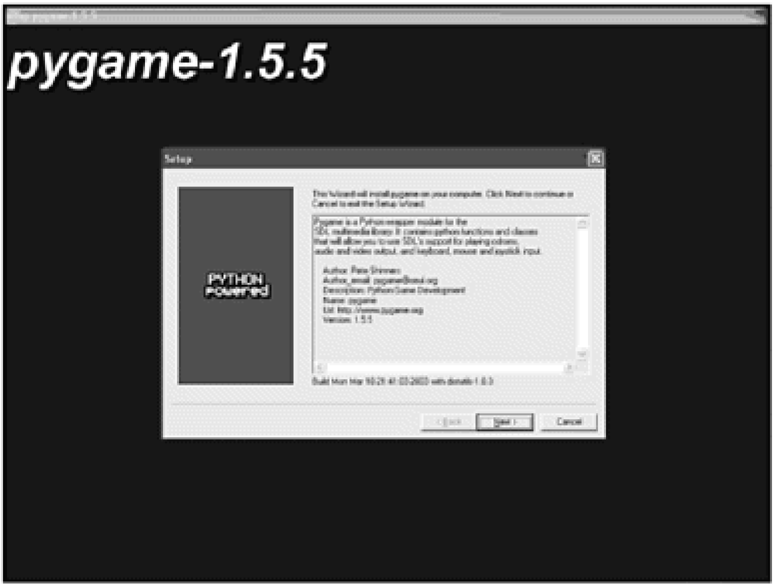
NOTE

CAUTION

Do not use stuffit to untar the package on Mac OS X. Stuffit will truncate some of the larger filenames.

Pygame is distributed under the GNU LGPL, license Version 2.1. See Figure 4.1 for a shot of Pygame installation.

Figure 4.1. Installing Pygame



Using Pygame

Pygame itself is fairly easy to learn, but the world of computer games and graphics is often unforgiving to beginners. Pygame has also suffered criticism for its lack of

documentation. This lack of documentation leads many new developers to browse through the Pygame package, looking for information. However, if you browse through the package, you will find an overwhelming number of classes at the top of the index, making the package seem confusing. The key to starting with Pygame is to realize that you can do a great deal with just a few functions, and that you may never need to use many of the classes.

Importing Pygame

The first step towards using Pygame after it has been installed is to import the Pygame and other modules needed for development into your code. Do the following:

```
import os, sys
import pygame
from pygame.locals import *
```

Keep in mind that Python code is case-sensitive, so for Python, Pygame and pygame are totally different creatures. Although I capitalize Pygame in this book's text, when importing the module, `pygame` needs to be in all lowercase letters.

First import a few non-Pygame modules. You'll use the os and sys libraries in the next few examples for creating platform independent files and paths. Then import Pygame itself. When Pygame is imported, it doesn't actually import all of the Pygame modules, as some are optional. One of these modules, called `locals`, contains a subset of Pygame with commonly used functions like `rect` and `quit` in the easy-to-access global namespace. For the upcoming examples the `locals` module will be included so that these functions will be available as well.

NOTE

TIP

The Pygame code repository is a community-supported library of tools and code that utilizes Pygame. The source code is managed by Pygame, but submissions are from users of the library. The repository holds a number of useful code snippets—everything from visual effects to common game algorithms—and can be found at http://www.pygame.org/pcr/.

The Pygame Surface

The most important element in Pygame is the surface. The surface is a blank slate, and is the space on which you place lines, images, color, and so on. A surface can be any size, and you can have any number of them. The display surface of the screen is set with:

```
Pygame.display.set_mode()
```

You can create surfaces that have images with `image.load()`, surfaces that contain text with `font.render()`, and blank surfaces with `Surface()`. There are also many surface functions, the most important being `blit()`, `fill()`, `set_at()`, and `get_at()`.

The `surface.convert()` command is used to convert file formats into pixel format; it sets a JPEG, GIF, or PNG graphic to individual colors at individual pixel locations.

NOTE

TIP

Using surface.convert is important so that SDL doesn't need to convert pixel formats on-the-fly. Converting all of the graphic images into an SDL format on-the-fly will cause a big hit to speed and performance.

Loading a surface image is fairly simple:

```
My_Surface = pygame.image.load('image.jpeg')
```

as is converting an image:

```
My_Surface =
pygame.image.load('image.jpeg').convert()
```

A conversion only needs to be done once per surface, and should increase the display speed dramatically.

Drawing on the display surface doesn't actually cause an image to appear on the screen. For displaying, the `pygame.display.update()` command is used. This command can update a window, the full screen, or certain areas of the screen. It has a counterpart command, `pygame.display.flip()`, which is used when using double-buffered hardware acceleration.

NOTE

CAUTION

The `convert()` command will actually rewrite an image's internal format. This is good for a game engine and displaying graphics, but not good if you are writing an image-conversion program or a program where you need to keep the original format of the image.

Creating a Pygame Window

Creating a window in which Pygame can run an application is fairly easy. First you need to start up Pygame with an initialize command:

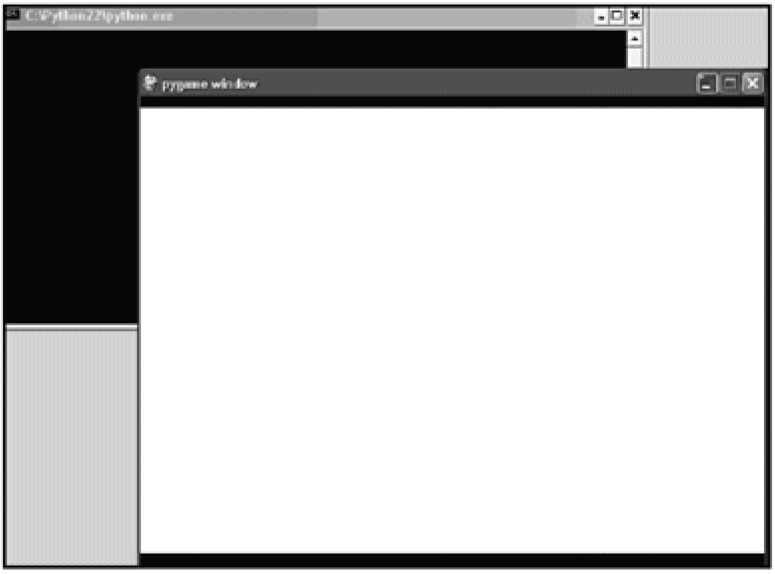
```
pygame.init()
```

Then you can set up a window with a caption by using Pygame's `display` command:

```
My_Window = pygame.display.set_mode((640, 480))
```

This code run by itself (the code is included as the `My_Window.py` example in this chapter's code section on the CD) creates a 640x480-pixel window labeled Pygame Window, just like in Figure 4.2. Of course, the window accomplishes nothing, so it immediately disappears after showing up on the screen.

Figure 4.2. A simple Pygame window



The Ever-Important rect()

The most used class in Pygame probably the `rect()` class, and it is the second most important concept in `Pygame. rect()` is a class that renders a rectangle:

```
My_Rectangle  = pygame.rect()
```

`rect()` comes with utility functions to move, shrink, and inflate itself; find a union between itself and other rects; and detect collisions. This makes `rect()` an ideal class for a game object. The position of a `rect()` is defined by its upper-left corner. The code that `rect`s use to detect overlapping pixels is very optimized, so you will see `rect`s used in sprite and other sorts of collision detection. For each object, there will often be a small `rect()` underneath to detect collisions.

The Event System

In order for Pygame to respond to a player or user event, you normally set up a loop or event queue to handle incoming requests (mouse clicks or key presses). This loop is a main `while` loop that checks and makes sure that the player is still playing the game:

```
still_playing = 1
while (still_playing==1):
        for event in pygame.event.get():
                if event.type is QUIT:
                        still_playing = 0
```

The `for event` line uses `pygame.event.get()` to get input from the user. Pygame understands basic Windows commands, and knows that `QUIT` is equivalent to pressing the X at the top right corner of a created window. The `pygame.event` function is used to handle anything that needs to go into the event queue—which is basically input from any sort of device, be it keyboard, mouse, or joystick. This function basically creates a new event object that goes into the queue. The `pygame.event.get` function gets events from the queue. The event members for `pygame.event` are

- **QUIT.** Quit or Close button.
- **ACTIVEEVENT.** Contains state or gain.
- **KEYDOWN.** Unicode key when pressed.
- **KEYUP.** Uncode key when released.
- **MOUSEMOTION.** Mouse position.
- **MOUSEBUTTONUP.** Position mouse button releases.
- **MOUSEBUTTONDOWN.** Position mouse button pressed.
- **JOYAXISMOTION.** Joystick axis motion.
- **JOYBALLMOTION.** Trackball motion.
- **JOYHATMOTION.** Joystick motion.
- **JOYBUTTONUP.** Joystick button release.
- **JOYBUTTONDOWN.** Joystick button press.
- **VIDEORESIZE.** Window or video resize.
- **VIDEOEXPOSE.** Window or video expose.
- **USEREVENT.** Coded user event.

These are normally used to track keyboard, mouse, and joystick actions. Let's say you wanted to build in mouse input handling. All mouse input is retrieved through the `pygame.event` module.

```
if event.type is MOUSEBUTTONDOWN:
        # do something
```

Pygame also has a number of methods to help it deal with actual mouse position and use; these are listed in Table 4.1.

Table 4.1. Pygame Mouse Event Methods

Method	Purpose
`get_cursor`	Gets the mouse cursor data

Table 4.1. Pygame Mouse Event Methods

Method	**Purpose**
`get_focused`	Gets the state of the mouse input focus
`get_pos`	Gets the cursor position
`get_pressed`	Gets the state of the mouse buttons
`get_rel`	Grabbing mouse movement
`set_cursor`	Sets the state of the shape of the mouse cursor
`set_pos`	Moves the cursor
`set_visible`	Displays or hides the mouse cursor

You can check the state of a mouse or keyboard event by using `pygame.mouse.get_pos()` or `pygame.key.get_pressed()`, respectively.

Drawing with Pygame

Pygame has great built-in functions for graphics. These functions revolve around the idea of the surface, which is basically an area that can be drawn upon. Let's say you wanted to fill the background in `My_Window.py` with a color. First grab the size of the window:

```
My_Background = pygame.Surface(My_Window.get_size())
```

This creates a surface called `My_Background` that's the exact size of `My_Window. Next` convert the surface to a pixel format that Pygame can play with:

```
My_Background = My_Background.convert()
```

And finally, fill the background with a color (set with three RGB values):

```
My_Background.fill((220,220,80))
```

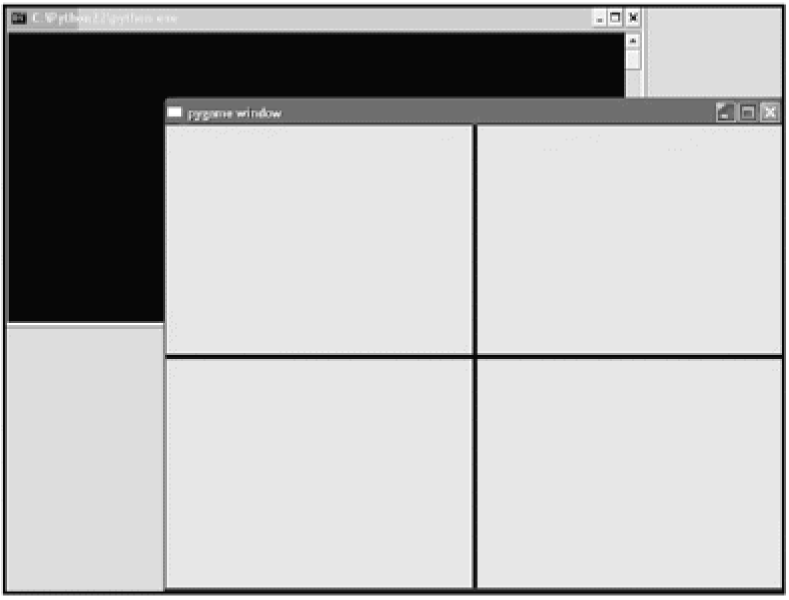
Now let's do some drawing over the background surface. Pygame comes with a `draw` function and a `line` method, so if you wanted to draw a few lines, you could do this:

```
pygame.draw.line (My_Background, (0,0,0,),(0,240),(640,240), 5)
pygame.draw.line (My_Background, (0,0,0), (320,0), (320,480), 5)
```

Pygame's `draw.line` takes five parameters. The first is the surface to draw on, the second is what color to draw (again in RGB values), and the last is the pixel width of the line. The middle parameters are the start and end points of the line in x and y pixel

coordinates. In this case, you draw the thick lines crossing in the exact center of the window, as shown in Figure 4.3.

Figure 4.3. Pygame's `draw.line` is used to split `My_Window` into four sections



The easiest way to display the background and lines is to put them into a `draw` function:

```
def draw_stuff(My_Window):
My_Background = pygame.Surface(My_Window.get_size())
My_Background = My_Background.convert()
My_Background.fill((220,220,80))
pygame.draw.line (My_Background, (0,0,0,),(0,240),(640,240), 5)
        pygame.draw.line (My_Background, (0,0,0), (320,0), (320,480),
5)
return My_Background
```

Then call the function within the loop that exists (and illustrated as code sample `My_Window_3.py` on the CD):

```
My_Display = draw_stuff(My_Window)
My_Window.blit(My_Display, (0,0))
pygame.display.flip()
```

Blitting

Blitting (Block Image Transfering) is practically synonymous with rendering, and specifically means redrawing an object by copying the pixels of said object onto a screen or background. If you didn't run the `blit()` method, nothing would ever get redrawn and the screen would just remain blank. For those of you who must know, `blit` isn't a made-up word—it's short for "bit block transfer."

In any game, blitting is often a process that slows things down, and paying attention to what you are blitting, when you are blitting, and how often you are blitting will have a major impact on your game's performance. The key to a speedy graphics engine is blitting only when necessary.

The `blit` method is very important in Pygame graphics. It is used to copy pixels from a source to a display. In this case, `blit` takes the pixels plotted in `My_Display` (which took the commands from `draw_stuff`) and copies them to `My_Window`. The `blit` method understands special modes like colorkeys and alpha, it can use hardware support if available, and it can also carry three-dimensional objects in the form of an array (using `blit_array()`). In this example, `blit` is taking `My_Display` as the input and rendering it to `My_Window`, and it uses the upper-left corner (pixel 0,0) to key up the surface.

The `pygame.display.flip()` command is Pygame's built-in function for updating the entire display (in this case, the entirety of `My_Window`) once any graphic changes are made to it.

NOTE

TIP

In Windows, you can add a single "w" to the end of a Python file (so that instead of ending it in py, it ends in pyw) to make the program open up a window without opening up the interpret console, that funny-looking DOS box.

Loading an Image with Pygame

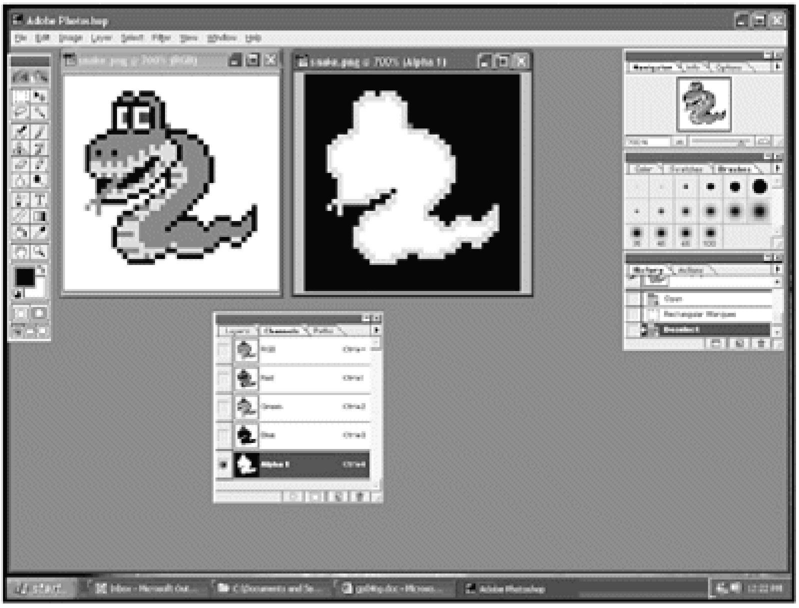
Image loading is an oft-needed function in games; in this section I'll show you the steps for loading an image in Pygame.

After importing the necessary modules, you need to define a function for loading an image that will take an argument. The argument will be used to set the colorkey (the transparency color) of the image; it looks like this:

```
def load_image(name, colorkey=None):
```

Colorkey blitting involves telling Pygame that all pixels of a specific color in an image should be transparent. This way, the image square doesn't block the background. Colorkey blitting is one way to make non-rectangular, two-dimensional shapes in Pygame. The other common trick is to set alpha values using a graphics program like Adobe Photoshop, as illustrated in Figure 4.4 and explained in the following sidebar.

Figure 4.4. Setting alpha values using Adobe Photoshop



To turn colorkey blitting on, you simply use `surface.set_colorkey(color)`. The color fed to `surface.set_colorkey` is three-digit tuple (0,0,0) with the first number being the red value, the second green, and the third blue (that is, rgb).

NOTE

Colorkey versus Alpha

Both colorkey and alpha are techniques for making parts of a graphic transparent when traveling across the screen. In Pygame, most 2D game objects and sprites are `rect`s, and are rectangular in shape. This means you need a way to make part of the rectangle transparent, so that you can have circular, triangular, or monkey-shaped game pieces. Otherwise you would only be capable of displaying square pieces over a background.

Alpha is one technique for making parts of an image transparent. An alpha setting causes the source image to be translucent or partially opaque. Alpha is normally measured from 0 to 255, and the higher the number is the more transparent the pixel or image is. Alpha is very easy to set in a graphic editor (like Adobe Photoshop), and Pygame has a built-in `get_alpha()` command. There is also per-pixel alpha where you can assign alpha values to each individual pixel in a given image.

When using a colorkey technique (sometimes called colorkey blitting) you let the image renderer know that all pixels of one certain color are to be set as transparent. Pygame has a built-in `colorkey(color)` function that takes in a tuple in the form of RGB. For instance, `set_colorkey(0,0,0)` would make every black pixel in a given image transparent.

You'll use both techniques in this chapter. The `load_image` function in this section uses `set_colorkey()`, while the `load_image` command in the Monkey_Toss.py graphics example later on in the chapter uses `get_alpha`.

The module needs to know where to grab the image, and this is where the os module comes into play. You'll use the os `path` function to create a full pathname to the image that needs to be loaded. For this example, say that the image is located in a "data" subdirectory, and then use the `os.path.join` function to create a pathname on whatever system (Mac, Windows, UNIX) that Python is running on.

```
fullname = os.path.join('data', name)
```

Try/except Code Blocks

Being able to fail gracefully is important in programming. Basically, you always need to leave a back door, or way out of a program, for if an error occurs. You'll find that `try/except` or `try/finally` constructs are very common.

Python offers a `try/except/else` construct that allows developers to trap different types of errors and then execute appropriate exception-handling code. `try/except` actually looks just like a series of `if/elif/else` program flow commands:

```
try:
        execute this block
except error1:
        execute this block if "error1" is generated
except error2:
        execute this block if "error2" is generated
else:
        execute this block
```

This structure basically allows for the execution of different code blocks depending on the type of error that is generated. When Python encounters code wrapped within a `try-except- else` block, it first attempts to execute the code within the `try` block. If this code works without any exceptions being generated, Python then checks to see if an `else` block is present. If it is, that code is executed.

If a problem is encountered while running the code within the `try` block, Python stops execution of the `try` block at that point and begins checking each `except` block to see if there is a handler for the problem. If a handler is found, the code within the appropriate `except` block is executed. Otherwise, Python jumps to the parent `try` block, if one exists, or to the default handler (which terminates the program).

A `try/except` structure is used to load the actual image using Pygame's image.load. Do this through a `try/except` block of code in case there is an error when loading the image:

```
try:
        image=pygame.image.load(fullname)
except pygame.error, message:
                print 'Cannot load image:', name
                raise SystemExit, message
```

Once the image is loaded, it should be converted. This means that the image is copied to a Pygame surface and its color and depth are altered to match the display. This is done so that loading the image to the screen will happen as quickly as possible:

```
image=image.convert()
```

The next step is to set the `colorkey` for the image. This can be the `colorkey` provided when the function was called, or a `-1`. If the `-1` is called, the value of `colorkey` is set to the top-left (0,0) pixel. Pygame's `colorkey` expects an RGBA value, and `RLEACCEL` is a flag used to designate an image that will not change over time. You use it in this case because it will help the speed of the image being displayed, particularly if the image must move quickly.

```
if colorkey is not None:
        if colorkey is -1:
                colorkey = image.get_at((0,0))
        image.set_colorkey(colorkey, RLEACCEL)
```

The final step is to return the `image` object as a `rect` (Like I've said, Pygame is based on `rect`s and surfaces) for the program to use:

```
return image, image.get_rect()
```

The full code snip for `load_image` is listed here and also on the CD, as `Load_Image.py`:

```
def load_image(name, colorkey=None):
fullname = os.path.join('data', name)
        try:
                image=pygame.image.load(fullname)
        except pygame.error, message:
                print 'Cannot load:', name
                raise SystemExit, message
        image=image.convert()
        if colorkey is not None:
                if colorkey is -1:
                        colorkey = image.get_at((0,0))
                image.set_colorkey(colorkey, RLEACCEL)
        return image, image.get_rect()
```

Displaying Text

Pygame has, of course, methods for dealing with text. The `pygame.font` method allows you to set various font information attributes:

```
My_Font = pygame.font.Font(None, 36)
```

In this case, you set up a `My_Font` variable to hold `Font(None, 36)`, which establishes no particular font type (`None`, which will cause a default font to be displayed) and a 36 font size (`36`). Step 2 is to choose what font to display using `font.render`:

```
My_Text = font.render("Font Sample", 1, (20, 20, 220))
```

The arguments passed to `font.render` include the text to be displayed, whether the text should be anti-aliased (1 for yes, 0 for no), and the RGB values to determine the text's color. The third step is to place the text in Pygame's most useful `rect()`:

```
My_Rect = My_Text.get_rect()
```

Finally, you get the center of both `rect()`s you created and the background with Python's super-special `centerx` method (which is simply a method for determining the exact center of something), and then call the `blit()` method to update:

```
My_Rect.centerx = My_Background.get_rect().centerx
background.blit(My_Text, My_Rect)
```

A Pygame Game Loop

A Pygame game loop is usually very straightforward. After loading modules and defining variables and functions, you just need a loop that looks at user input and then updates graphics. This can be done with only a few lines of code. A typical event loop in a game would look something like this:

```
while 1:
        for event in pygame.event.get():
                if event.type == QUIT:
                        #exit or quit function goes here
                        return
                screen.blit(MY_Window, (0, 0))
                pygame.display.flip()
```

The `pygame.event` module looks for user input, and `pygame.blit` and `pygame.display` keep the graphics going. Let's say, for example, that you wanted to look specifically for up or down arrow keys for player control. To do so, you could simply add `elif` statements to the event loop:

```
while 1:
        for event in pygame.event.get():
                if event.type == QUIT:
                        #exit or quit function goes here
                        return
                # Add to listening for arrow keys In the event queue
                elif event.type == KEYDOWN:
                        If event.key == K_UP
                                # do something
                        If event.key == K_DOWN
                                # do something
```

```
screen.blit(MY_Window, (0, 0))
pygame.display.flip()
```

Pygame Sprites

Originally computers were simply incapable of drawing and erasing normal graphics fast enough to display in real-time for purpose of a video game. In order for games to work, special hardware was developed to quickly update small graphical objects, using a variety of special techniques and video buffers. These objects were dubbed sprites. Today sprite usually refers to any animated two-dimensional game object.

Sprites were introduced into Pygame with Version 1.3, and the `sprite` module is designed to help programmers make and control high-level game objects. The `sprite` module has a base class `Sprite`, from which all sprites should be derived, and several different types of `Group` classes, which are used as Sprite containers.

When you create a sprite you assign it to a group or list of groups, and Pygame instantiates the `sprite` game object. The sprite can be moved, its methods can be called, and it can be added or removed from other groups. When the sprite no longer belongs to any groups, Pygame cleans up the `sprite` object for deletion (alternately, you can delete the sprite manually using the `kill()` method).

The `Group` class has a number of great built-in methods for dealing with any sprites it owns, the most important being `update()`, which will update all sprites within the group. Several other useful group methods are listed in Table 4.2.

Table 4.2. Useful Group Methods

Method	**Use**
`add()`	Adds a sprite to the group
`copy()`	Makes a copy of the group with all of its members
`empty()`	Removes all sprites within the group
`len()`	Returns how many sprites the group contains
`remove()`	Removes sprite from the group
`truth()`	Returns true if group has any sprites
`update()`	Calls an update method on each sprite within the group

Groups of sprites are very useful for tracking game objects. For instance, in an asteroid game, player ships could be one group of sprites, asteroids could be a second group, and enemy starships a third group. Grouping in this way can make it easy to manage, alter, and update the sprites in your game code.

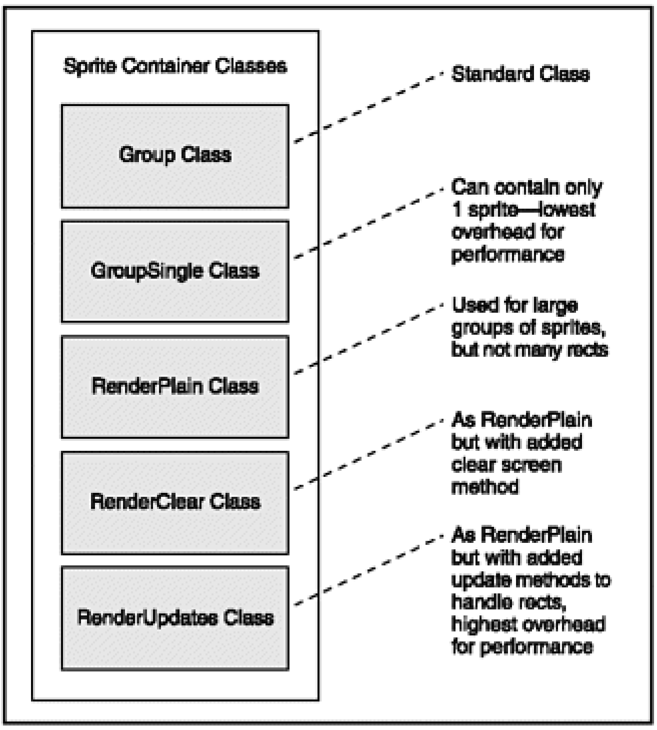
Memory and speed are the main reasons for using sprites. Group and sprite code has been optimized to make using and updating sprites very fast and low-memory

processes. Pygame also automatically handles cleanly removing and deleting any `sprite` objects that no longer belong to any groups.

Updating an entire screen each time something changes can cause the frames-per-second rate to dip pretty low. Instead of updating the entire screen and redrawing the entire screen normally, an engine should only change the graphics that have actually changed or moved. The engine does this by keeping track of which areas have changed in a list and then only updating those at the end of each frame or engine cycle. To help out in this process, Pygame has different types of groups for rendering. These methods may not work with a smooth-scrolling, three-dimensional, realtime engine, but then again, not every game requires a whopping frame-rate. Pygame's strength lies elsewhere.

Besides the standard `Group` class there is also a `GroupSingle`, a `RenderPlain`, a `RenderClear`, and a `RenderUpdates` class (see Figure 4.5). `GroupSingle` can only contain one sprite at any time. Whenever a sprite is added to `GroupSingle`, any existing sprite is forgotten and set for deletion. `RenderPlain` is used for drawing or blitting a large group of sprites to the screen. It has a specific `draw()` method that tracks sprites that have `image` and `rect` attributes.`RenderPlain` is a good choice as a display engine for a game that scrolls through many backgrounds but not any `rects`, like scrolling games where the player stays in a consistent area of the screen and the background scrolls by to simulate movement. `RenderClear` has all the functionality of `RenderPlain` but also has an added `clear()` method that uses a background to cover and erase the areas where sprites used to reside. `RenderUpdates` has all the functionality of `RenderClear`, and is also capable of tracking any `rect` (not just sprites with `rect` attributes) for rendering with `draw()`.

Figure 4.5. Sprite container classes



Sprites also have built-in collision detection. The `spritecollide()` method checks for collisions between a single sprite and sprites within a specific group, and will return a list of all objects that overlap with the sprite if asked to. It also comes with an optional `dokill` flag, which, if set to true, will call the `kill()` method on all the sprites.

A `groupcollide()` method checks the collision of all sprites between two groups and will return a dictionary of all colliding sprites if asked to. Finally, the `spritecollideany()` method returns any single sprite that collides with a given sprite. The structure of these collision methods is:

```
pygame.sprite.spritecollide(sprite, group, kill?) ->list
pygame.sprite.groupcollide(group1, group2, killgroup1?, killgroup2?) -
> dictionary
pygame.sprite.spritecollideany(sprite, group) -> sprite
```

Here is an example of a collision that checks to see whether `My_Sprite` ever collides with `My_Player`, and removes the offending `My_Sprite` sprite:

```
for My_Sprite in sprite.spritecollide(My_Player, My_Sprite, 1):
        #What happens during the collision plays out here
```

When using Pygame sprites, you need to keep a few things in mind. First, all sprites need to have a `rect()` attribute in order to use the `collide()` or most other built-in methods. Second, when you call the Sprite base class to derive your sprite, you must call the `sprite_init_()` method from your own `class_init_()` method.

Game Object Classes

Python being a pseudo–object-oriented language, normally game classes are created first, then specific instances of game objects are initiated from the created classes. Let's walk through creating an example class, a banana:

```
class Banana:
        # _init_ method
# banana method
        # banana method 2
        # banana method 3
def main
        My_Banana = Banana()
```

This is roughly how a class works. The `Banana` class needs at least an `_init_` method, and will likely contain many more. After the class is created, simply call the class to create an instance called `My_Banana` in the main loop.

Since an `_init_` method is mandatory, let's take a look at what that method would look like first:

```
class Banana(pygame.sprite.Sprite):
        def _init_(self):
                pygame.sprite.Sprite._Init_(self)
```

The `Banana` class is set up as a Pygame sprite. When you define the `_init_` method, you must specify at least one parameter that represents the object of the class for which the method is called. By convention, this reference argument is called `self`.

You may want to add other specifications to the `_init_` method. For instance, you may wish to specify an `image/rect` and load up a graphic. You may also want to tie the `Banana` class to the screen:

```
class Banana(pygame.sprite.Sprite):
        def _init_(self):
                pygame.sprite.Sprite._Init(self)
                self.Image, self.rect = load_png('banana.png')
                screen = pygame.display.get_surface()
```

After defining `_init_`, you may also want to add methods that define the object's position on the screen, and update the object when necessary:

```
class Banana(pygame.sprite.Sprite):
        def _init_(self):
                pygame.sprite.Sprite._Init(self)
                self.Image, self.rect = load_png('banana.png')
                screen = pygame.display.get_surface()
        def Bannana_Position(self, rect)
                # Funky math here
                # that defines position on screen
                return position
        def Banana_Update(self)
                # Code that updates the banana
```

Pygame Drawbacks

Pygame is simply a wrapper around SDL, which is a wrapper around operating system graphic calls. Although programming is much easier when using Pygame than when using SDL, Pygame removes you pretty far from the code that actually does the work, and this can be limiting in a number of ways.

Probably the most significant drawback to Pygame, however, is the fact that the library needs so many dependencies in order to function. Obviously, Pygame needs Python and SDL to run, but it also needs several smaller libraries, including `SDL_ttf`, `SDL_mixer`, `SDL_image`, `SDL_rotozoom`, and the Python Numeric package for the `surfarray` module. Some of these libraries have their own dependencies.

UNIX packages generally come with package and dependency managers that make managing dependencies a controllable problem in UNIX. But on Windows systems, it can be difficult to distribute a game without creating a collection of all the needed files the game requires to run.

Luckily, there are Python tools to help build Windows executables. I mentioned a few of these in Chapter 3, in particular a tool called Py2exe. Pete Shinners, the Pygame author, actually wrote a tutorial on how to use Py2exe to package a Python Pygame for Windows. The tutorial comes with a sample distutils script and can be found at http://www.pygame.org/docs/tut/Executable.html.

Finally, although hardware acceleration is possible with Pygame and fairly reliable under Windows, it can be problematic because it only works on some platforms, only works full screen, and greatly complicates pixel surfaces. You also can't be absolutely sure that the engine will be faster with hardware acceleration—at least not until you've run benchmark tests.

A Pygame Example

In this section you'll use the Pygame `load_image` function with game loops to create a simple two-dimensional graphics-engine game example. The steps you need to take are as follows:

1. Import the necessary libraries.
2. Define any necessary functions (such as `load_image`).
3. Define any game object classes (sprites, game characters).
4. Create a main event loop that listens for events.
5. Set up Pygame, the window, and the background.
6. Draw and update necessary graphics (utilizing groups and sprites).

I envision a monkey-versus-snakes game, where the monkey/player throws bananas at snakes to keep them at bay. The steps for coding this example are explained in each of the following sections, the full source code can be found on the CD as `Monkey_Toss.py`, and Figure 4.6 gives you a preview of the game.

Figure 4.6. A preview of Monkey_Toss.py



Importing the Necessary Libraries

Importing has been covered ad nauseum already, so I will not bore you with the details. Simply start with this code:

```
# Step 1 - importing the necessary libraries
import pygame, os
import random
from pygame.locals import *
```

These libraries should be familiar to you with the exception of the `random` module. Python comes equipped with `random`, and we will be using the `random.randrange` method to generate random numbers.

NOTE

Random Library

The `random.randrange` method generates a random number (an integer) within the range given. For instance, this snippet prints a number between 1 and 9:

```
import random
Print ( random.randrange(1, 10))
```

Simple enough. Note that `random.randrange` prints up to the highest number given, but not the actual highest digit. Random numbers are used so often in games that you will often encounter random number functions like this:

```
Def  DiceRoll():
      Dice1 = random.randrange( 1, 7)
      Print "You rolled %d" % (dice1)
      Return dice1
```

You will be using random's `randrange()` and `seed()` methods to produce random numbers for the `Monkey_Toss.py` example.

Defining Necessary Functions

You will be using a version of `load_image` in this game example, but you will switch from using `colorkey` and look instead for alpha values in the graphics. You have the graphics already built with alpha channels and stored in a data directory next to the game code (and also on the CD). This means you need to alter a few lines of code from `Load_Image.py`:

```
def load_image(name):
    fullname = os.path.join('data', name)
    try:
        image = pygame.image.load(fullname)
        # Here instead of the colorkey code we check for alpha values
        if image.get_alpha is None:
                image = image.convert()
        else:
                image = image.convert_alpha()
    except pygame.error, message:
        print 'Cannot load image:', fullname
        raise SystemExit, message
    return image, image.get_rect()
```

You will also define a very short function to help handle keystroke events from the player. We will call this function `AllKeysUp`:

```
def AllKeysUp(key): return key.type == KEYUP
```

Defining Game Object Classes

First you will define a sprite class. The class needs, of course, an `_init_` method:

```
class SimpleSprite(pygame.sprite.Sprite):
        def __init__(self, name=None):
                pygame.sprite.Sprite.__init__(self)
                if name:
                        self.image, self.rect = load_image(name)
                else:
                        pass
```

When initiating, you set `SimpleSprite` to load the given image name and become a `rect()`. Normally, you would include error code in case the name isn't passed or something else goes wrong, but for now you will just use Python's `pass` command (`pass` is an empty statement that can be used for just such a situation).

You will also give your `SimpleSprite` a method to set up its surface:

```
        def set_image(self, newSurface, newRect=None):
                self.image = newSurface
                if newRect:
                        self.rect  = newRect
                else:
                        pass
```

Normally you would set up each pass default and also include at least a base method for updating the sprite, but for now let's keep it easy.

For this engine, as I said, I envisioned a monkey versus snakes game, and since you are writing in Python, start with the `Snake_Grass` class:

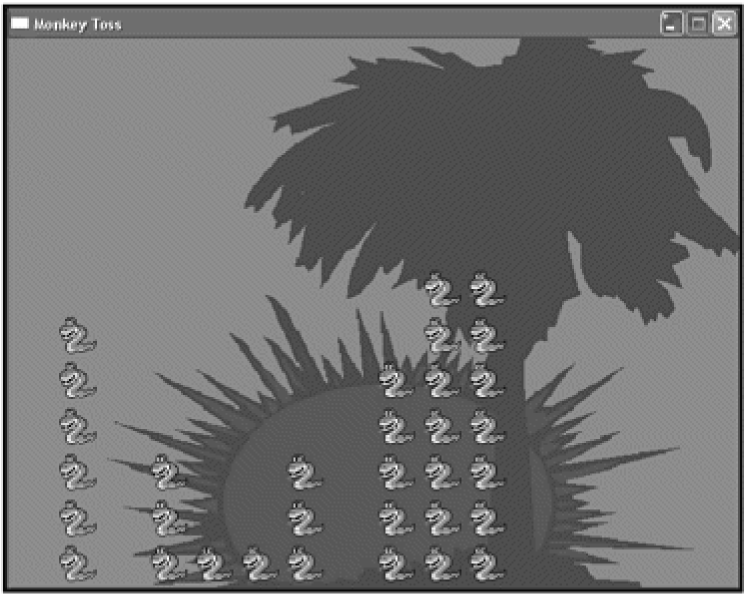
```
class Snake_Grass:
    def __init__(self, difficulty):
        global snakesprites
        global block
        for i in range(10):
            for j in range(random.randrange(0,difficulty*5)):
                block = SimpleSprite("snake.png")
                block.rect.move_ip(((i+1)*40),480-(j*40))
                snakesprites.add(block)

    def clear(self):
        global snakesprites
        snakesprites.empty()
```

There are two methods in this class, one to initiate the object and one to clear it. The `clear()` method simply uses `empty()` to clear out the global `snakesprites` when it is time. The `_init_` method takes in the required `self` and also a measure of difficulty, ensures `snakesprites` and `block` are created, and then starts iterating through a `for` loop.

The outer `for` loop iterates through a second inner `for` loop that creates a random number of "blocks," each of which contains a square `snakesprites`loaded with the snake.png graphic. These sprites are created and moved into stacks on the game board using a bit of confusing math (`block.rect.move_ip(((i+1)*40),480-(j*40))`). Don't worry too much about the math that places these sprites on your 480 pixel-wide surface; instead, realize that when initiated with an integer representing difficulty, a `Snake_Grass` object will create a playing board similar to that in Figure 4.7.

Figure . Figure 4.7 `Snake_Grass`object called with a difficulty of 2



The placement of the `snakesprites` and the height of the rows are random so that a differ ent game board surface is produced each time the game is run.

Define the player sprite next; this will be `Monkey_Sprite`. You want the `Monkey_Sprite` to possess the ability move in the game, so you need to define a number of methods to define and track movement:

```
class Monkey_Sprite(pygame.sprite.Sprite):
    def __init__(self, game):
    # For creating an Instance of the sprite
    def update(self):
    # Update self when necessary
    def check_crash(self):
    # Check for collision with other sprites
    def move(self):
    # How to move
    def signal_key( self, event, remainingEvents ):
    # Respond to player If they me to do something
    def check_land(self):
    # See If reach the bottom of the screen
```

That's a lot of methods, but in actuality, the `Monkey_Sprite` is fairly uncomplicated once you take the time to walk through each method. Lets start with `_init_`:

```
def __init__(self, game):
         pygame.sprite.Sprite.__init__(self)
         self.image, self.rect = load_image('monkey.png')
        self.rightFacingImg = self.image
        self.leftFacingImg = pygame.transform.flip( self.image, 1, 0)
         self.direction = 0
         self.increment = 25
         self.oldPos = self.rect
         self.game = game
         self.listenKeys = {}
```

First you load the image into a `rect()` that will represent the `Monkey_Sprite` game object, monkey.png, on the game board surface. Then you set a number of variables. The `rightFacingImg` is the normal state of the graphic, and the `leftFacingImg` is the graphic rotated 180 degrees using the Pygame's handy `transform.flip()` method.

The `self.direction` value is a Boolean value that will either have the `Monkey_Sprite` traveling left (represented by a 0) or right (represented by a 1). Set `self.increment` to 25, representing 25 pixels that the `Monkey_Sprite` will travel with each update. The next three settings are all set for the methods that follow and use them.

`Update` is the next method:

```
    def update(self):
        self.check_land()
        self.move()
        if self.direction == 0:
            self.image = self.rightFacingImg
        else:
            self.image = self.leftFacingImg
        self.check_crash()
```

`Update` first checks, using the `check_land` method, to see whether the `Monkey_Sprite` has reached the bottom of the screen. You haven't defined `check_land` yet, but you will momentarily. Then `update` moves the `Monkey_Sprite` with the `move` method, which you also have yet to define. It then checks which direction `Monkey_Sprite` is facing and makes sure the graphic being used is facing the correct way. Finally, `update` calls `check_crash`, which also needs to be defined, and checks to see whether there have been any sprite collisions.

The `check_land` method simply looks to see if the `Monkey_ Sprite` has crossed a particular pixel boundary on the game board surface, which is defined by the `self.rect.top` and `self.rect.left` variables. If it has, then we know that the `Monkey_Sprite` needs to start back over at the top of the screen.

```
    def check_land(self):
        if (self.rect.top == 640) and (self.rect.left == 1):
            self.game.land()
```

The `move` method uses the defined increment value you set in `_init_` to move the sprite across the screen in the direction you've set. If the sprite goes outside the game window (>640 or <0 pixels), you make the sprite switch and travel back across the screen in the opposite direction:

```
def move(self):
         self.oldPos = self.rect
         self.rect = self.rect.move(self.increment, 0)
         if self.rect.right > 640:
             self.rect.top += 40
             self.increment = -25
```

```
            self.direction = 1
        if self.rect.left < 0:
            self.rect.top += 40
            self.increment = 25
            self.direction = 0
```

The `check_crash` method uses Pygame's built-in group methods and `pygame.sprite.spritecollide()` to check if the `Monkey_Sprite` ever collides with anything in the crash list, which in this case includes any `snakesprites`. If there is a crash, `Monkey_Sprite` will call the `game.crash()` method, which we will define momentarily.

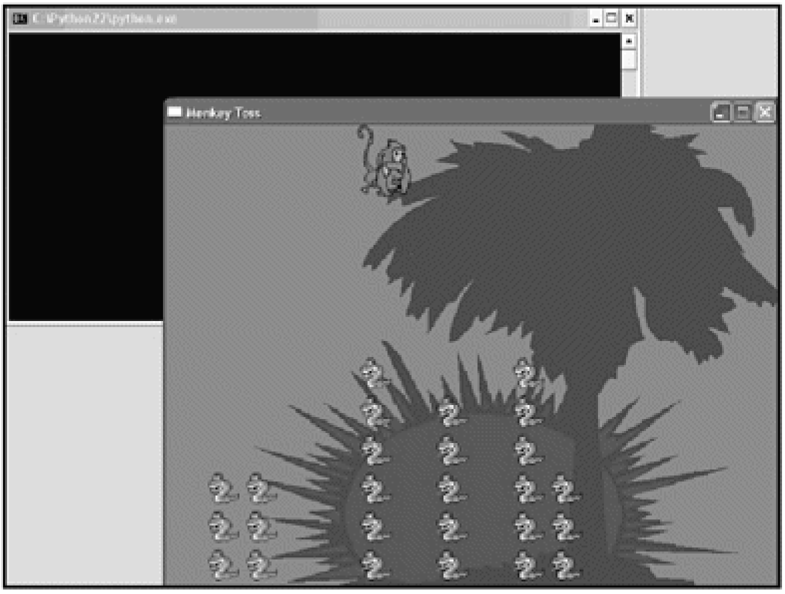
```
def check_crash(self):
        global snakesprites
        crash_list = pygame.sprite.spritecollide(self, snakesprites, 
0)
        if len(crash_list) is not 0:
            self.game.crash(self)
```

Only one more method is associated with the `Monkey_ Sprite`, `signal_key`, which is simply a listener for keyboard events.

```
def signal_key( self, event, remainingEvents ):
                if self.listenKeys.has_key( event.key ) \
                  and event.type is KEYDOWN:
                        self.listenKeys[event.key]( remainingEvents )
```

Once a `MonkeySprite` object is loaded, it will appear in the top-left corner of the game board surface and travel across the screen, as shown in Figure 4.8. When it hits the edge of the screen, it drops a little and then heads back in the opposite direction. If the `Monkey_Sprite` ever touches a `snakesprite` or the bottom of the screen, he will start back at the top again.

Figure 4.8. An instance of the `Monkey_Sprite` class travels across the screen



Now you have monkeys and snakes. You need one more actor, a banana, which the `Monkey_Sprite` objects will throw at and destroy the snake objects with. This means you need methods for the banana to update and move and check for collisions:

```
class Banana(pygame.sprite.Sprite):
    def __init__(self, rect, game):
    def update(self):
    def move(self):
    def check_hit(self):
```

Initializing the banana sprite works much like the other `_init_` methods. There will be an incremental value that defines how many pixels the banana moves when updated, and the sprite that represents the banana will load up a `rect()` and fill it with the fruit.png file. Finally, you will need some code to check with the master `game` object for when the banana collides or moves off the screen:

```
def __init__(self, rect, game):
        pygame.sprite.Sprite.__init__(self)
        self.increment =16
        self.image, self.rect = load_image("fruit.png")
        if rect is not None:
            self.rect = rect
        self.game = game
```

Updating and moving are also set up like the other classes. The banana moves according to its increment value and checks are required to see if the banana collides with any sprites or moves off of the game board surface:

```
def update(self):
        self.move()
    def move(self):
        self.rect = self.rect.move(0, self.increment)
```

```
        if self.rect.top==480:
            self.game.miss()
        else:
            self.check_hit()
```

Finally, the `check_hit` method looks for any collisions with `snakesprites` just like with the `Monkey_Sprite`:

```
    def check_hit(self):
        global snakesprites
        collide_list = pygame.sprite.spritecollide(self,
snakesprites,0)
        if len(collide_list) is not 0:
            self.game.hit()
```

There is still one more class to write—the most important and lengthy game object. You are actually going to put the game controls and variables into a game class called `MonkeyToss`. We need `MonkeyToss` to be able to handle a number of different things, but mostly keyboard events, collisions, and actions for when sprites move off the screen. This gives `MonkeyToss` several different methods:

```
class MonkeyToss:
    def __init__(self, charGroup):
    def crash(self, oldPlane):
    def land(self):
    def drop_fruit(self):
    def miss(self):
    def signal_key( self, event, remainingEvents ):
    def hit(self):
```

The master game class initializes pretty much everything else you need as far as game mechanics. First, it takes in the game sprites and assigns them to the `charGroup` group. Then it defines the game difficulty that the rest of the classes use. The specifc keybard key the sprite needs to respond to is the spacebar, which when pressed will fire the `drop_fruit` method. Finally the snake, monkey, and banana (`fruit`) are all initialized:

```
def __init__(self, charGroup):
         self.charGroup = charGroup
         self.difficulty = 2
         self.listenKeys = {K_SPACE: self.drop_fruit}
         self.snake = Snake_Grass(self.difficulty)
         self.monkey = Monkey_Sprite(self)
         self.charGroup.add( [self.plane] )
         self.fruit = None
```

The `crash` method is called by our `Monkey_Sprite` when it collides with a `snakesprite`. When the `Monkey_Sprite` collides with a `snakesprite`, it needs to be destroyed with the `kill()` method and then a new `Monkey_Sprite` should be instantiated to start over and be assigned to the sprite group:

```
def crash(self, oldMonkey):
```

```
        self.monkey.kill()
        self.monkey = Monkey_Sprite(self)
        self.charGroup.add ([self.monkey])
```

The `land` method is also called by the `Monkey_Sprite` when it reaches the bottom of the screen. For this sample the method is identical to the `crash` method, but in a real game, the landing might create a new field of snakes, or pop the player to a different area of the game entirely.

```
def land(self):
        self.monkey.kill()
        self.monkey = Monkey_Sprite(self)
        self.charGroup.add([self.monkey])
```

The `drop_fruit` method is called when the spacebar is pressed, and `Monkey_Sprite` attempts to drop fruit on a snake. `Drop_fruit` assigns `self.fruit` an instance of the Banana class and adds it to the active sprite group:

```
    def drop_fruit(self):
        if self.fruit is None:
            self.fruit = Banana(self.monkey.rect, self)
            self.charGroup.add([self.fruit])
```

Code must be created for when the dropped fruit falls past the end of the screen; for our purposes the sprite can just call the `kill()` method on itself:

```
def miss(self):
        self.fruit.kill()
        self.fruit = None
```

For keyboard events, define a `signal_key` method:

```
    def signal_key( self, event, remainingEvents ):
        if self.listenKeys.has_key( event.key ):
            self.listenKeys[event.key]()
        else:
            self.monkey.signal_key( event, remainingEvents )
```

The last part is the code that handles sprite collision. This bit is fairly complex. First you need to keep track of all the `snakesprites`, and then all of the sprites in the group, by creating `My_Group`. Then you call `colliderects[]`, which returns true if any `rect` in the group collides:

```
def hit(self):
        global snakesprites
        My_Group = pygame.sprite.Group()
        colliderects = []
```

Following `colliderects[]` is a for loop that basically checks to see if the bottom of the fruit `rect` and the top of the monkey `rect` collide, and if so adds them to the `collide` list:

```
        for i in range(3):
            for j in range((self.fruit.rect.bottom+16-
self.monkey.rect.top)/16):
                rect = Rect((self.fruit.rect.left-32+i*32,
self.fruit.rect.
bottom-j*16),(25,16))
                colliderects.append(rect)
```

Then, for each collision, you need to destroy the given `fruit` and make sure the sprite group is updated:

```
        for rect in colliderects:
            sprite = SimpleSprite()
            sprite.rect = rect
            My_Group.add(sprite)
        list = pygame.sprite.groupcollide(My_Group, snakesprites,
1,1)
        self.fruit.kill()
        self.fruit = None
```

That's quite a lot of work, but, happily, defining the classes comprises the bulk of this sample's code, and you are past the halfway point of coding. Now onwards!

Creating a Main Event Loop that Listens for Events

To create a main loop, you normally define a main function containing a `while` loop:

```
def main():

        while 1:
               # do stuff
if __name__ == "__main__":
    main()
```

This ensures that `main()` is called and your `while` loop keeps running during the course of the game. As good coding practice, initialize a few variables inside of `main()`:

```
    global screen
    global background
    global snakesprites
    global block
```

You are also going to take advantage of a Pygame `clock` feature and use random's `seed` method to set a random number seed. Since you are going to be experiencing movement and time, you'll be setting an `oldfps` variable to help keep track of time and loop iterations:

```
    clock = pygame.time.Clock()
    random.seed(111111)
    oldfps = 0
```

Finally, the `while` loop. You want to make sure time is recorded by using `clocktick()` and updating with each iteration. Any keyboard events are queued, so that QUIT, the Escape key, or the `KEYUP`, which is set to be the Spacebar, can be responded to:

```
while 1:
        clock.tick(10)
        newfps = int(clock.get_fps())
        if newfps is not oldfps:
            oldfps = newfps
        oldEvents = []
        remainingEvents = pygame.event.get()
        for event in remainingEvents:
            oldEvents.append( remainingEvents.pop(0) )
            upKeys = filter( AllKeysUp, remainingEvents )
            if event.type == QUIT:
                return
            elif event.type == KEYDOWN and event.key == K_ESCAPE:
                return
            elif event.type == KEYDOWN or event.type == KEYUP:
                game.signal_key( event, upKeys )
```

Setting Up Pygame, the Window, and the Background

You can initialize Pygame using the `init()` method within `main()`. Then you use `display.set_mode()` to configure the game surface to 640x480 pixels, and the game caption to be "Monkey Toss". You then use your `load_image` method to load up the surface background and initialize blitting and flipping:

```
    pygame.init()
    screen = pygame.display.set_mode((640, 480))
    pygame.display.set_caption('Monkey Toss')
    background, tmp_rect = load_image('background.png')
    screen.blit(background, (0, 0))
    pygame.display.flip()
```

Drawing and Updating Necessary Graphics

For drawing, you start by initializing all of your sprites and sprite groups in `main()`:

```
    allsprites = pygame.sprite.RenderUpdates()
    snakesprites= pygame.sprite.RenderUpdates()
    block = None
    game = MonkeyToss(allsprites)
```

The code that does all the work lies at the end of the `while` loop, which clears the sprite groups then updates and redraws each changed `rect()`:

```
allsprites.clear( screen, background)
snakesprites.clear(screen, background)
allsprites.update()
changedRects2 = allsprites.draw(screen)
changedRects3 = snakesprites.draw(screen)
pygame.display.update(changedRects2+changedRects3)
```

The finished product and the full source code and data files can be found in Chapter 4's file on the CD. Obviously, quite a bit could be added to this program. Check out the complete game sample at the end of this chapter for a few ideas!

Python Graphics

Choosing a graphics toolkit may be the most difficult choice when creating a game. There are hundreds of graphic kits to choose from and each is very different in style and language. This chapter only covers a handful of the graphics libraries available for Python programming, and goes through samples in only a few of the available options—mainly the popular kits available for developing cross-platform.

Specifically, more coverage of Tkinter is given in this section, as Tkinter comes bundled with Python, is cross platform, and is commonly used as a GUI for Python programs. Pygame is probably the most popular Python game library in use today, and Pygame graphic calls have already been covered in some detail. A few OpenGL samples in Python are also examined at the end of this chapter.

NOTE

A number of commercial art tools are programmable with in Python scripts. Some of the more recognizable tools include Blender, Poser, Lightflow, and Softimage XSI. Each of these tools has a Python interface. Blender (i.e. gameBlender) uses Python as a scripting language, the Poser Pro pack includes a Python-scripting agent, Lightflow has a Python extension module, and Softimage is scriptable via Python.

For the aspiring developer, there are also many other graphic options available. Here, for starters, is a short list of Python GUI libraries and graphics kits:

- **The Standard Window Interface.** STDWIN used to be the most commonly used GUI for Python, but is now largely unsupported. The library was meant to be a platform-independent interface to C-based Windows systems, but the module no longer exists in Python 2.0 or above, and I mention it mainly for legacy. It runs under UNIX and Mac, but was never ported to Windows.
- **The Wxpython library.** Provides support for the wxWindows-portable GUI class library. Wxpythin uses the Lesser Gnu Public License and functions like a wrapper to the C++ wxWindows library. It is relatively cross platform, but not quite as portable as Tkinter.
- **The Pythonwin library.** Pythonwin is also included in many standard Python distributions, but applications designed with it will only run on Windows. Pythonwin is a wrapper to the Microsoft Foundation Class Library, and provides features of the Windows user interface.
- **Wpy.** An object-oriented, cross-platform class library system also based on the Microsoft Foundation Classes. Wpy is built to be simple and portable.
- **PyKDE.** A set of Python bindings for the KDE classes written by Phil Thompson. PyKDE requires Sip to run.
- **PyGTK.** A free software GUI toolkit that has a large number of widgets oriented towards the X Window System. PyGTK is distributed under the Lesser Gnu Public License and was developed for the GTK widget and GNOME libraries. The library is object-oriented and comes with lots of good samples.
- **GNOME Python.** A set of bindings for the GNOME libraries that use PyGTK (which comes bundled with the package).
- **Wafepython.** Wafe is short for Widget Athena Front End, and is a package for developing applications with high-level graphical user interfaces in Tcl.

WafePython implements an interface between Tcl, the X Toolkit, the Athena Widget Set, the Motif Widget Set, and a few other classes and widget packages thrown in for good measure.

- **PyFLTK.** FLTK stands for Fast Light Toolkit; it's a C++ GUI toolkit for UNIX, OpenGL and Win32. PyFLTK was originally created to build in-house apps for Digital Domain. Bill Spitzak is the original author and received permission from the company to distribute it under the Lesser Gnu Public License. Other developers have done more work on the toolkit since then, and the project has been moved to Sourceforge.
- **Fox Python.** FXPy is a C++ toolkit for developing GUIs that runs on UNIX and Windows; it is distributed under the Lesser Gnu Public License. Fox's emphasis is on speed and ease of use. It uses techniques for increasing drawing speed and minimizing memory, and most controls can be built with a single line of code. Fox supports drag and drop, OpenGL widgets, 3D graphics, and tooltips.
- **Python X.** An extension that binds Python together with Motif, which is a set of user interface guidelines set by the Open Software Foundation. Motif is actually over a decade old, and there are many books covering its use, but it has been somewhat in decline for a while.
- **The Python Computer Graphics Kit.** A collection of Python modules for 3D computer-graphics images. The kit mainly focuses on Pixar's RenderMan interface, but some modules can also be used for OpenGL programs or non-RenderMan-compliant renderers.
- **Vpython.** A free and open-source 3D programming library designed "for ordinary mortals." The idea behind Vpython is ease of use and simplicity.
- **Zoe.** A bare-bones OpenGL graphics engine written completely in Python. Zoe includes only basic 3D features, and focuses on creating 3D wire-frames for prototyping or rapid development.
- **The PyUI Library.** An interface library written entirely in Python for Python. It can run on desktop Windows or in a 3D hardware-accelerated environment and is meant to be portable. PyUI was originally slated to build user interfaces for games. PyUI is owned by Sean Riley of Ninjaneering (see Chapter 5 for more information on Ninjaneering) and utilizes Python 2.1, Pygame, PyOpenGL, the Python Imaging Library, and the ActiveState win32 extensions.
- **PyQT.** Qt for Windows is a C++ cross-platform GUI toolkit distributed by TrollTech, who have a free non-commercial version license and a pay commercial license. PyQT is a set of Python bindings to the C++ QT Toolkit, originally produced by the Kompany and now under River Bank Computing. The GUI toolkit runs on Windows, Mac OS X, and UNIX.

NOTE

TIP

GUIs are created with graphical elements called widgets, which are typically scrollbars, buttons, text fields, etc. Widgets are normally found within a window, which controls the layout of the widgets.

Python also has a few basic built-in tools for graphics and image handling. These are included under its Multimedia Services modules, which are listed in Table 4.3.

Table 4.3. Python Multimedia Graphic Services

Module	Use
`colorsys`	Converting between RGB and other color systems
`imageop`	Manipulating raw image data
`imghdr`	Determining the type of image contained in a file or bytestream
`rgbimg`	Reading and writing image files in SGI RGB format

The `imageop` module can operate on 8- or 32-bit pixel images and has methods for cropping, scaling, dithering, and converting the image at a raw level. `Colorsys` can be used to convert RGB, HLS, HSV, and YIQ color systems. Python's `imghdr` can recognize a number of different image formats (as shown in Table 4.4) and is also extendable to allow even more types.

Table 4.4. Image Formats

Value	Image format
rgb	SGI ImgLib Files
gif	GIF 87a and 89a Files
pbm	Portable Bitmap Files
pgm	Portable Graymap Files
ppm	Portable Pixmap Files
tiff	TIFF Files
rast	Sun Raster Files
xbm	X Bitmap Files
jpeg	JPEG data in JFIF Format
bmp	BMP Files
png	Portable Network Graphics

The Tkinter Library

In the last chapter you built a small display box using Tkinter. Here you'll explore GUI creation with Tkinter in more depth. As you recall, Tkinter is an object-oriented interface that works on multiple platforms and is designed to be extensible so that it can be used to import third-party widgets.

Widgets

Tkinter comes with only a handful of standard widgets. Each widget has a standard set of methods and also supports a large set of general methods, so they are capable of a wide coverage. There is a lot more to widgets than what's listed in Chapter 3 (reprinted here as Table 4.5 for easy reference). This is because each of these components has its own place and use within a GUI, and therefore has its own components and methods associated with it.

Table 4.5. Tkinter Widget Components

Component	Function
`Button`	Creates a button that triggers an event when clicked
`Canvas`	Displays text or images
`Checkbutton`	Creates a Boolean checkbutton
`Entry`	Creates a line that accepts keyboard input
`Frame`	Creates the outlying window's edge
`Label`	Displays text as labels for components
`Listbox`	Creates a list of options
`Menu`	Creates a multiple-selection display
`Menubutton`	Creates a pop-up or pull-down style menu
`Radiobutton`	Creates a single option button
`Scale`	Creates a slider that can choose from a range
`Scrollbar`	Creates a scrollbar for other components
`Text`	Creates a multiple-line box that accepts user input
`Toplevel`	A widget container like `Frame` but with its own top-level window

NOTE

Tcl/TK

TK is a toolkit that handles the creation of windows, GUI events (widgets), and user interaction. The TK toolkit is provided as an extension for Tcl. Tkinter is an interface to Tcl; without the interface it would take hundreds of lines of code to do even simple things like open a window or create a button.

Many languages use or are capable of using TK. Tkinter is Python's behind-the-scenes director of the TK GUI toolkit, and Tcl is the behind-the-scenes director that Tkinter uses to communicate to TK. Both TK and Tcl are open-source developments that are under development at scriptics (the Tcl developer exchange can be found at http://dev.scriptics.com).

Button

Clickable buttons are probably the most widely used widget in any interface, and Tkinter has a many options available for button components; these are listed in Table 4.6.

Table 4.6. Button Properties

Property	**Function**
`activebackground`	Sets the background color
`activeforeground`	Sets the foreground color
`bitmap`	Displays a given bitmap as the button
`default`	Identifies the default button
`disabledforeground`	Sets a foreground color used when button is disabled (grayed out)
`image`	Sets an image to display in the widget (precedes bitmap)
`state`	Defines the button state (as `NORMAL`, `ACTIVE`, or `DISABLED`)
`takefocus`	Indicates whether the Tab key can be used to reach this button
`text`	Defines the text to display within the button
`underline`	An offset applied on text displayed to identify which character must be underlined
`wraplength`	Determines distance when text should be wrapped to the next line

Buttons also have their own special methods: `flash()` is a method which reverses and resets the foreground and background attributes, and `invoke()` is a method that executes the function defined in a command.

I used a button widget in the last chapter's GUI sample, inititated by the following code and looking like Figure 4.8 (a short `Hello_Button.py` sample is also given in this chapter's code section on the CD):

```
Button(window, text='Exit', command=window.quit).pack(side=BOTTOM)
```

This can be broken down into basic components. `Button()` is used to create the button, and the parameters placed within the `Button()` parentheses, (`window, text='Exit', command=win dow.quit`), define what the button can do. The `pack()` method extends `Button()` and defines where the button should be placed within the window, in this case `side=BOTTOM`.

Canvas

The Canvas widget component is used to draw everything from arcs to bitmaps to polygons. It is used as a way to customize graphical items, and resembles an artist's

blank canvas, ready to be painted. A canvas in Tkinter, of course, has its own properties; these are listed in Table 4.7.

`Hello_Canvas.py` is given on the CD as a sample that produces a large widget surface, as shown in Figure 4.10.

Figure 4.10. Sample Canvas widget

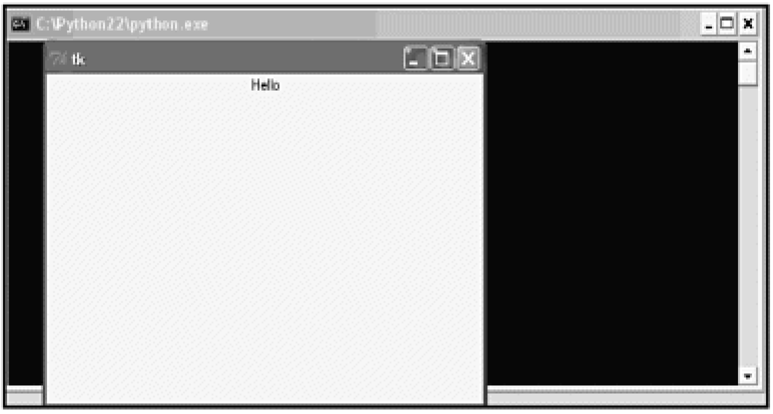


Figure 4.9. The widget at work



Table 4.7. Canvas Properties

Property	Function
`arc`	Creates an arc or an arc item
`bitmap`	Creates a bitmap item
`image`	Creates an image item
`line`	Creates a line item
`oval`	Creates a circle or ellipse at the given coordinates
`polygon`	Creates a polygon item (three or more vertices) with the given coordinates

Table 4.7. Canvas Properties

Property	Function
`rectangle`	Creates a rectangle item with the given coordinates
`text`	Creates a text item at the given position with the given options
`window`	Embeds a window widget to the canvas

Checkbutton

A Checkbutton is basically a box that can either be checked or unchecked; an example is shown in Figure 4.11 and a sample is included in the CD's source code as `Hello_Checkbutton.py`. Checkbuttons can have an on value and an off value set for whether the box is checked, and have a handful of methods available, as shown in Table 4.8.

Figure 4.11. A sample checkbutton



Table 4.8. Checkbutton Methods

Method	Function
`select()`	Selects the checkbutton and sets the value of the variable to `onvalue`
`flash()`	Reverses and resets the foreground/background colors
`invoke()`	Executes a function defined by `command()`
`toggle()`	Reverses the state of a button (i.e. off becomes on)

Entry

The Entry widget is designed to let users enter a single line of text within a frame or window. A sample `Hello_Entry.py` is included on the CD.

Frame

A Frame widget is used to group, arrange, and organize other widgets. It uses rectangular screen areas and padding to put them into view for a GUI. A sample `Hello_Frame.py` is included on the CD.

Label

A Label widget is a box that displays text or images. The Label widget allows you to create and update these displays, and a demonstration is given as `Hello_Label.py` on the CD.

Listbox

A Listbox widget creates lists of text items that can be selected by the user. Listboxes have three properties:

- **height.** The number of rows in the list. Setting `height` to 0 allows `listbox` to automatically resize to the number of entries.
- **selectmode.** Defines the type of list being created. This can be `SINGLE`, `EXTENDED`, `MULTIPLE`, or `BROWSE`.
- **width.** The Number of characters in each row, which can also be automatically resized with the setting 0.

The Listbox widget also has a number of methods associated with it, as shown in Table 4.9.

Table 4.9. Listbox Methods

Method	Function
`delete()`	Deletes a given row, or the rows between the given row and `lastrow`
`get()`	Gets the string that starts at the given row
`insert()`	Inserts the given string at the given row
`see()`	Makes the row visible to the user
`select_clear()`	Clears the selection
`select_set()`	Selects the rows starting at `startrow` and ending at `endrow`

A Listbox example is on the CD as `Hello_Listbox.py`.

Menu

There are three types of Menu widgets: pop-up, toplevel, and pull-down. There are also special menu widget item types such as radio menu items and check menu items. A sample menu is given as `Hello_Menu.py`. Menus, of course, have their own methods, as listed in Table 4.10:

Table 4.10. Menu Methods

Methods	Function
`add_command()`	Adds a menu item
`add_radiobutton()`	Creates a radio button menu item
`add_checkbutton()`	Creates a check button menu item
`add_cascade()`	Creates a new hierarchical menu
`add_separator()`	Adds a separator line to the menu
`add()`	Adds a specified type of menu item
`delete()`	Deletes the menu items from `startindex` to `endindex`
`entryconfig()`	Modifies a menu item
`index()`	Returns the index number to the given menu item

These methods have their very own options available to them, as shown in Table 4.11.

Menubutton

Menubuttons can be used to display menus, but are in decline since the Menu widget has been expanded to include most of the Menubutton functionality.

Message

Message is very similar to the Label widget, and is used to create a multiple line non - editable object that displays text.

Radiobutton

Radio button widgets are multiple-choice buttons. Each group of radio buttons must be associated to the same variable, and each Radiobutton must represent a single value at any given time. Radiobuttons have their own properties:

- **command.** Function to be called when the button is clicked.
- **variable.** Variable to updated when button is clicked.
- **value.** Defines the value that is stored in the variable when button is clicked.

Table 4.11. Menu Widget Method Options

Option	Function
`accelerator`	A keyboard alternative to a menu option
`command`	Names the `callback` function when the menu item is selected
`indicatorOn`	Adds a switch next to the menu options

Table 4.11. Menu Widget Method Options

Option	Function
`label`	Defines the text of the menu items
`selectColor`	Switches color (with `indicatorOn`)
`state`	Defines menu item status (`normal`, `active`, or `disabled`)
`onvalue`	Values to be stored in the variable property
`offvalue`	Values to be stored in the variable property
`tearOff`	Creates a clickable separator at the top of the menu
`underline`	Defines the index position of the character to be underlined
`variable`	Variable used to store a value

Radiobuttons also have their own special methods:

- **flash().** Reverses forground and background colors.
- **invoke().** Executes command function.
- **select().** Selects the radio button.

A Radiobutton is shown in Figure 4.12 and a sample is included in the CD samples as `Hello_Radiobutton.py`.

Figure 4.12. A radio button



Scale

A scale widget is a graphical slider object that allows a user to select values from a scale. Scale has its own unique methods:

- **get().** Gets the current scale value.
- **set().** Sets the scale to a specified value.

`Hello_Scale.py` is included on the CD as a sample and Figure 4.13 displays the output of the sample code.

Figure 4.13. A Scale sample widget



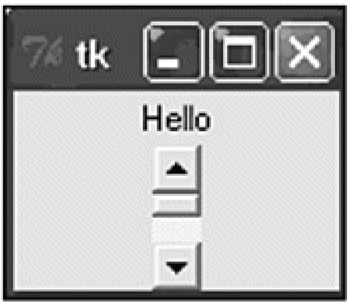
Scrollbar

A scrollbar widget is used to select from a vertical scroller and works with `listbox`, `text`, and `canvas`. Scrollbar in Tkinter has the same methods available as `scale`:

- **set().** Defines fractions between 0 and 1 that delimit the view.
- **get().** Returns the current scrollbar configuration settings.

A sample scrollbar is incuded on the CD (`Hello_Scrollbar`) and also illustrated in Figure 4.14.

Figure 4.14. A scale sample widget



Text

Text allows the editing and formatting of multiple lines of text and has a number of available methods, as listed in Table 4.12.

Table 4.12. Text Methods

Method	Function
`delete()`	Deletes specified character(s)
`get()`	Returns specific character(s)
`index()`	Returns absolute value of an index
`insert()`	Inserts string at a specified index
`see()`	Returns true if the text located at a given index is visible

There are also a few available attributes for text:

- **state.** Sets text to editable or non-editable with the flags `normal` or `disabled`.
- **tabs.** Provides a list of strings and identifies table stops on the Text widget.

Text widgets support bookmark positions, called Marks; the naming of regions of texts, called Tabs; and specific locations, called Indexes, to help them organize text. Each of these three—Marks, Tabs, and Locations—has access to specified methods.

Toplevel

The Toplevel widgets are directly managed by the window manager; its methods are listed in Table 4.13.

Universal Widget Methods

All widgets in Tkinter also have standard universal options for defining things they have in common. They all use a similar syntax, and are listed in Table 4.14.

There are also methods inherited from the base Tk classes that are provided for all Tkinter widgets, including the toplevel object created by the `Tk()` method. These always apply to the widget that makes the method call, and are listed in Table 4.15. Take notice of the idea of focus with these methods. The window or widget that is in focus is the one that is toplevel to the viewer.

Table 4.13. Toplevel Methods

Method	Function
`aspect()`	Controls the relation between height and width
`client()`	Used in X windows to define `WM_CLIENT_MACHINE`
`colormapwindows()`	Used in X windows to define `WM_COLORMAP_WINDOWS`
`command()`	In X defines `WM_COMMAND`
`deiconify()`	Displays the window

Table 4.13. Toplevel Methods

Method	Function
`frame()`	Returns the window identifier
`focusmodel()`	Sets the focus model
`geopmetry()`	Changes the window's geometry
`group()`	Adds given window to the window group
`iconbitmap()`	Defines a bitmap for when the window is iconified
`iconify()`	Turns the window into an icon
`iconmask()`	Defines an icon bitmap for when the window is iconified
`iconname()`	Defines an icon name for when the window is iconified
`iconposition()`	Defines a suggestion for where the icon goes when the window is iconified
`iconwindow()`	Defines the icon window that should be used as an icon
`maxsize()`	Defines the maximum size for the window
`minsize()`	Defines the minimum size for the window
`overrideredirect()`	Defines a flag different from 0, and tells the window manager not to add a title or borders to the window
`positionfrom()`	Defines the position controller
`protocol()`	Registers a function with a callback
`resizable()`	Defines resize flags
`sizefrom()`	Defines size controller
`state()`	Returns the current state of the window, being normal, iconic, withdrawn, or icon
`title()`	Defines the window title
`transient()`	Turns window into a temporary window for the given master which is automatically hidden
`withdrawn()`	Removes the window from the screen

Table 4.14. Standard Tkinter Widget Options

Standard Widget Option	Properties
`height`	Defines height in number of characters or pixels
`width`	Defines width in pixels or number of characters
`background or bg`	Defines background color
`foreground or fg`	Defines foreground color

Table 4.14. Standard Tkinter Widget Options

Standard Widget Option	Properties
`relief`	Defines border style
`highlightcolor`	Defines color used to draw the highlight region when widget has keyboard focus
`highlightbackground`	Defines color used to draw the highlight region when widget does not have keyboard focus
`highlightthickness`	Defines highlight region width in pixels
`borderwidth or bd`	Width of widget relief border in pixels
`text`	Contains widget caption text, formatted by foreground and font
`justify`	Sets `LEFT`, `RIGHT`, or `CENTER` for text captions
`font`	Can define font family, font size, and font values like bold, underline, and overstrike
`command`	Associates a widget with a Python function
`variable`	Maps widget to a variable
`anchor`	Defines location of a widget within a window or of text within a widget
`padx`	Defines padding on the x-axis to border
`pady`	Defines the padding on the y-axis to border
`cursor`	Defines mouse pointer when moved over widget

NOTE

CAUTION

Colors can vary from platform to platform. For instance, the Windows operating system has system color settings for windows in the Control Panel, while the UNIX X Window System keeps them in an xrgb text file. This could cause GUI color choices to change slightly (or radically) from one operating system to the next.

Table 4.15. Tkinter Widget Methods

Method	Function
`cget()`	Returns a string that contains the current configuration value for a given option
`config()`	Sets the values for one or more options
`configure()`	Same as `config()`

Table 4.15. Tkinter Widget Methods

Method	Function
`destroy()`	Destroys the widget
`focus()`	Sets the widget to a keyboard focus
`focus_set()`	As `focus()`
`focus_display()`	Returns the name of the window that contains the widget and has focus
`focus_force()`	Gives keyboard focus to the widget
`focus_get()`	Returns the identity of the window that has focus
`focus_lastfor()`	Returns the window that last had focus
`getvar()`	Returns the value of a Tkinter variable
`grab_set()`	Grabs all events for the entire screen for the widget
`grab_release()`	Releases grab on a widget
`grab_set_global()`	Returns `none`, `local`, or `global` depending upon the grab value set to a window
`keys()`	Returns all options available for a widget as a tuple
`lift()`	Moves a widget to the top of the window stack
`tkraise()`	Same as `lift()`
`lower()`	Moves a widget to the bottom of the windows stack
`mainloop()`	Activates the `mainloop` event
`quit()`	Quits the `mainloop` event
`setvar()`	Sets a value to a given Tkinter variable
`update()`	Processes all queued tasks
`update_idletasks()`	Processes all pending idle tasks
`tk_focusNext()`	Returns the next widget that should have keyboard focus
`tk_focusPrev()`	Returns the previous widget that should have keyboard focus
`wait_variable()`	Creates a local event that waits for the given Tkinter variable to change
`wait_visibility()`	Creates a local event that waits for the given widget to become visible
`wait_window()`	Creates a local event that waits for a given widget to be destroyed

There are also specific methods for all widgets that work within windows. For ease of reference, they begin with a `winfo` (short for Window Information). These methods are listed in Table 4.16.

Table 4.16. Widget Window Information Methods

Method	Function
`winfo_cells()`	Returns the number of cells in the widgets color map
`winfo_children()`	Returns a list of widget instances
`winfo_class()`	Returns the Tkinter class name for widget
`winfo_colormapfull()`	Returns true if a widget's colormap is full
`winfor_containing()`	Returns the identity of the widget at the given x + y coordinates
`winfo_depth()`	Returns bit depth of the widget (8, 16, 24, or 32 bits per pixel)
`winfo_exists()`	Returns true if a Tk window corresponds to the given widget
`winfo_fpixels()`	Returns the result of the conversion of the given distance to the corresponding number of pixels (in floating point value)
`winfo_geometry()`	Returns a string showing the widget coordination in pixels
`winfo_height()`	Returns pixel height
`winfo_width()`	Returns pixel width
`winfo_id()`	Returns window identity
`winfo_ismapped()`	Returns true if a widget is mapped by the window system
`winfo_manager()`	Returns the name of the geometry manager
`winfo_name()`	Returns widget name
`winfo_parent()`	Returns widget parent
`winfo_pathname()`	Returns pathname of widget
`winfo_pixels()`	Same as `winfo_fpixels()` except returns a regular integer instead of a floating point value
`winfo_pointerx()`	Returns the x coordinate of the mouse pointer in pixels (must be in widget window)
`winfo_pointery()`	Returns the y coordinate of the mouse pointer in pixels (must be in widget window)
`winfo_reqheight()`	Returns minimum height required by widget to be displayed
`winfo_reqwidth()`	Returns minimum width required by widget to be displayed
`winfo_rootx()`	Returns the pixel coordinates of a widget's upper-left corner
`winfo_rooty()`	Returns the pixel coordinates of a widget's upper-left corner

Table 4.16. Widget Window Information Methods

Method	Function
`winfo_screen()`	Returns the screen name for the current window
`winfo_screencells()`	Returns the number of cells in the default color map for widget's screen
`winfo_screendepth()`	Returns the bit depth of the window target
`winfo_screenheight()`	Returns the height of a widget screen in pixels
`winfor_screenwidth()`	Returns width of widget screen in pixels
`winfo_screenmmheight()`	Returns screen height but in millimeters
`winfo_screenmmwidth()`	Returns screen width but in millimeters
`winfo_screenvisual()`	Returns the default visual class used for widget's screen (i.e. grayscale, truecolor, staticcolor, and so on)
`winfo_toplevel()`	Returns the widget instance of the top-level window containing the widget
`winfo_visual()`	Returns the visual class used for the widget (grayscale, truecolor, staticcolor, etc.)
`winfo_x()`	Returns x axis pixel coordinates corresponding to the widget's upper-left corner, relative to upper-left corner of the parent
`winfo_y()`	Returns y axis pixel coordinates corresponding to the widget's upper-left corner, relative to upper-left corner of parent

Tkinter Geometry

Tkinter widgets have specific geometry management methods that are used to organize widgets in their area. These methods are organized in three classes that help a UI designer develop an interface. The methods are `pack()`, `grid()`, and `place()`.

Using these methods is fairly effortless. First you create a widget. In the last chapter you created a widget frame called `window`:

```
Import Tkinter *
window = frame()
```

After you have a widget, you can simply and easily apply `pack()`, `grid()`, or `place()` directly on it:

```
window.pack()
window.grid()
window.place()
```

Using these three methods is very important in organizing a GUI interface, so I'll cover each one in the next subsections.

pack()

The `pack()` method is used to organize widgets in blocks before placing them in the parent widget. `pack()` adds a widget to a frame or window based on the order that the widgets are packed. If you don't specify how the widgets are to be packed, they are simply placed top to bottom in the available space. You can, however, specify placement with options like `anchor` or `side`. The `pack()` method has a few built-in methods, shown in Table 4.17.

Table 4.17. *`pack()`* Method Options

Option	Use
`Expand`	Expands a widget to use up available space
`Fill`	Defines how a widget should fill a parcel or frame
`Ipadx`	Used with `fill` to define space in pixels around an object
`Ipady`	Used with `fill` to define space in pixels around an object
`Padx`	Defines space in pixels between widgets
`Pady`	Defines space in pixels between widgets
`Side`	Defines where you want to place the widget (chosen from `TOP`, `BOTTOM`, `LEFT`, and `RIGHT`)

NOTE

TIP

The default is to use pixels to define measurement in `pack()`, but you can define different measurements, such as onscreen centimeters (c), onscreen millimeters (m), inches (i), and printer points (p). You specify which measurement to use by adding the letters to the options measurements:

```
# this specifies padding to be in inches
window.pack(padx=4i, pady=5y)
```

grid()

The `grid()` method is used to organize widgets via a table within the parent widget. `grid()` creates a grid pattern (go figure) within a frame, and then allocates space to each cell in the grid to hold a widget. This grid starts are location (0,0) at the top left of the window. `Grid()` has a few methods, outlined in Table 4.18.

Table 4.18. `grid()` Method Options

Option	Use Example
`Column`	Specifies the column number
`Columnspan`	To make a widget span multiple (default is 1 column)
`Row`	Specifies the row number
`Rowspan`	To make a widget span multiple rows (default is one row)

place()

The `place()` method is used to place widgets in specific a specific position in the parent widget. `place()` allows you to set the exact position and size of each widget, in terms of absolute or relative coordinates. The `place()` method can use the options listed in Table 4.19.

Table 4.19. `place()` Method Options

Option	use
`Anchor`	Defines coordinates by (by compass: `N`, `S`, `E`, `W`, `NE`, `NW`, `SE`, `SW`, `or CENTER`). Default value is `NW`
`Bordermode`	Defines `INSIDE` or `OUTSIDE`
`Height`	Defines widget height in pixels
`In`	Places widget in a position relative to the given widget (`in_`)
`Relheight`	Defines relative height in reference to `in_`
`Relwidth`	Defines relative width in reference to `in_`
`Rely`	Defines relative position in, reference to `in_`
`Relx`	Defines relative position in reference to `in_`
`Width`	Defines widget width in pixels
`Y`	Define absolute position of widget on y-axis, default 0
`X`	Define absolute position of widget on x-axis, default 0

Tkinter Events

Events in Tkinter are user events like keyboard presses and mouse movements. Tkinter handles events by creating bindings for specific objects. You can bind events to a widget, to the widget's Toplevel window, to a widget's class, or to an entire application.

Once an event has been bound to a widget, you specify a callback, which is a function that is called when the event happens. Let's say you had a function called `My_Event`:

```
def My_Event():
        //does something here
```

Let's say you want `My_Event` to be called by a widget button called `My_Button`:

```
My_Button = Button()
```

The `My_Button` widget can call `My_Event` by simply including a command option on one line:

```
My_Button['command'] = My_Event
```

You can assign events to keyboards and mouse presses as well, as shown in Table 4.20 and Table 4.21.

Table 4.20. Tkinter Mouse Events

Event	Effect
`<Button -1>`	Mouse button (left) is pressed over widget
`<Button -2>`	Mouse button (middle) is pressed over widget
`<Button -3>`	Mouse button (right) is pressed over widget
`<B1 –Motion>`	Mouse is moved with the button held down (dragged)
`<ButtonRelease -1>`	Mouse button is released
`<Double – Button - 1>`	A double click
`<Enter>`	Mouse pointer enters widget
`<Leave>`	Mouse pointer leaves widget

Table 4.21. Tkinter Keyboard Events

Event	Effect
`<Alt -x>`	Pressed Alt and another key
`<Control -X>`	Pressed Ctrl and another key
`<Escape>`	Pressed the Esc key
`<key>`	Press any key (carries the character pressed via a callback)
`<Return>`	Pressed the Enter key
`<Shift -X>`	Pressed Shift and another key

The object that originated the callback exposes the attributes for events. These attributes are listed in Table 4.22.

Table 4.22. Tkinter Event Attributes

Object	Attribute
`Char`	Character code of pressed key
`Height`	New height of a widget in pixels
`Keycode`	Key code of a pressed key
`Keysym`	Key symbol of a pressed key
`Num`	The mouse button number associated with an event (usually 1, 2, or 3)
`Type`	The event type
`Widget`	The widget instance
`Width`	New width of a widget in pixels
`X`	The current position in pixels of the mouse on the x-axis
`X_root`	The current x-axis position of the mouse in pixels relative to the upper-left corner of the screen
`Y`	The current position in pixels of the mouse on the y-axis
`Y_root`	The current y-axis position of the mouse in pixels relative to the upper-left corner of the screen

NOTE

TIP

For Tkinter mouse events, you will often find `<Button -1 >` replaced with `<ButtonPress-1>` or `<1>`, all of which are correct syntactically. These changes work for the middle and right-side buttons as well.

For Tkinter keyboard events, most keys can be represented by placing them within less than and greater than symbols (`<F1>`, `<Cancel>`, and `<End>`, for example).

There are also methods used to handle a callback by binding a Python function or method to an action that can be applied to a widget. These are shown in Table 4.23.

Table 4.23. Tkinter Event Callbacks

Method	Event
`after()`	Alarm callback called after given time in milliseconds
`after_cancel()`	Cancels an alarm callback
`after_idle()`	When the system is idle, registers a callback
`bindtags()`	Returns the search order used by widget
`bind()`	Defines the callback that must be associated to a given event

Table 4.23. Tkinter Event Callbacks

Method	Event
`bind_all()`	Defines the callback that must be associated to a given event at the application level
`bind_class()`	Defines the callback that must be associated to a given event at the given widget class
`<Configure>`	Widget is resized or moved to a new location
`unbind()`	Removes bindings for the given event
`unbind_all()`	Removes bindings at the application level
`unbind_class()`	Removes bindings for the given event at the given widget class

Finally, Tkinter has protocols to handle events that communicate between the window manager and the GUI. This allows an application to intercept messages from the system and act accordingly. These protocols were original established for the X system, but Tk can handle events on multiple platforms. The syntax to bind a protocol to a handle event is as follows:

```
widget.protocol(protocol, handler)
```

In order for the widget to intercept a system message it needs to be on the Toplevel. The handler is almost always a function.

Tkinter Images

Tkinter uses the `image` class as a foundation to display graphic objects. Graphic objects Tkinter can display include both bitmap (`BitmapImage`) and GIF (`PhotoImage`) images. The functions `image_names` and `image_types` are used to handle all the `images` within the `image` class. The first returns a list containing the names of all available images, and the second returns a list that contains all the existing types that were created.

Images, once created, provide a handful of methods: `image.width()`, `image.type()`, and `image.height()`.

BitmapImage

`BitmapImage` is used to display bitmap images on widgets. In Tkinter, however, a bitmap not a .bmp format image. Bitmaps are actually two color images (well, two colors and a transparency mask to be precise) and have the options listed in Table 4.24.

Table 4.24. `BitmapImage` Options

Method	Purpose

Table 4.24. `BitmapImage` Options

Method	Purpose
`cget()`	Returns value of the given option
`config()`	Changes image options
`configure()`	Changes image options
`height()`	Returns height in pixels
`width()`	Returns width in pixels
`type()`	Returns the bitmap string

These options have methods available to them, listed in Table 4.25.

Table 4.25. `BitmapImage` Option Methods

Method	Used For
`background`	Background color
`data`	String to be used instead of a file
`file`	File to be read
`foreground`	Foreground color to be used
`format`	Specifies the file handler to be used
`maskdata`	String that defines the contents of the mask
`maskfile`	Specifies mask file
`height`	Gives image dimensions
`width`	Gives image dimensions

PhotoImage

`PhotoImage` is used for displaying full color images; it supports GIF and PPM files and has attributes as listed in Table 4.26.

Table 4.26. PhotoImage `Attributes`

Attribute	Holds
`data`	String to be used instead of a file
`file`	File to be read
`height`	Dimensions
`width`	Dimensions

The PyOpenGL Library

PyOpenGL is an OpenGL widget written by a large group of developers, including David Ascher, Mike Hartshorn, Jim Hugunin, and Tom Schwaller. PyOpenGL includes OpenGL bindings for Python created using the Simplified Wrapper and Interface Generator (SWIG) and distributed under open source licenses. It supports OpenGL v1.0, OpenGL v1.1, GLU, GLUT v3.7, GLE 3, WGL 4, and Togl (Tk OpenGL widget). PyOpenGL is also interoperable with Tkinter, wxPython, FxPy, PyGame, and Qt and a large number of other external GUI libraries for Python. It has a very active following and a regularly updated sourceforge project page at http://pyopengl.sourceforge.net/.

OpenGL has the reputation of being difficult to learn. Hey, there are reasons why they pay game developers the big bucks! Python's version of OpenGL is no different than any other version, and OpenGL looks pretty similar no matter what language you're playing with.

The reason OpenGL is considered difficult to pick up is because three-dimensional graphics programming can be a fairly difficult subject just on its own. Since OpenGL is fairly difficult to master, this section covers just a few examples. If you discover, as many programmers do, that OpenGL is your calling, then I recommend that you pick up OpenGL Game Programming by Kevin Hawkins and Dave Astle.

Using OpenGL in Python is quite an advantage over other languages, however, because Python and Pygame make several complex steps much easier. For instance, I use the python.game window in these examples to open up a window for displaying graphics. This could take dozens of lines of code in a non–high-level language, but it only takes two in these examples. You also do not have to worry about freeing and releasing memory for all of the complex graphics calls and routines. However, having no control over memory allocation and de-allocation can cause problems.

NOTE

OpenGL

OpenGL is a standard graphics library originally created by Silicon Graphics. Back then it was called GPL, and only ran on SGI hardware. SGI eventually turned their technology into an open standard and licensed it to different machines. OpenGL may be the premier development tool for developing portable 2d and 3d applications, and it has also been a standard since the early 1990s.

OpenGL is free for application and game designers. It is an owned technology, but the licensing applies to venders of hardware (i.e., the graphic card makers) that wish to utilize the technology, not the software developers. SGI is currently working towards modifying the license into a true open source license. This makes OpenGL very popular among game developers, and many commercial games have used it, from Activision's Quake, to Blizzard's Diablo, to Bioware's NeverWinter Nights.

Installing PyOpenGL

PyOpenGL needs a handful of dependencies in order to access all of its functionality. Luckily, most of these will already be installed if you've been playing with the code in this chapter. PyOpenGL needs Python 2.2 or higher, Tcl/Tk, OpenGL, GLU (which should come pre-installed on most modern machines and with most modern graphics card), the OpenGL Utility Toolkit (or GLUT for short), and Numeric Python.

The OpenGL Context may also require a few dependencies, depending on the platform. Those dependencies that are freely distributable are on this book's CD, under \PYTHON\PYOPENGL\DEPENDENCIES, except for Numeric Python, which has its own folder (\PYTHON\NUMERIC PYTHON). The standard binary installers for PyOpenGL are located on the CD under \PYTHON\PUOPENGL. The source and project page for PyOpenGL can be found at Sourceforge, which is where you will want to look for the latest updates and news: http://pyopengl.sourceforge.net/documentation/installation.html

Using PyOpenGL

There are four libraries to PyOpenGL, each of which is normally imported separately:

- **GL.** The basic, primitive library.
- **GLU.** Short for GL utilities; includes more advanced commands than GL.
- **GLX.** GL for `X_Windows`.
- **GLUT.** GL Utilities Toolkit, which has even more sophisticated windowing features.

For these samples you will be using both GL and GLU:

```
from OpenGL.GL import *
from OpenGL.GLU import *
```

To make things easier, you will also be using bits of the Pygame library:

```
import pygame
from pygame.locals import *
```

First a small program creates a PyOpenGL Window with a graphic on a Win32 platform. This first program, labeled `OpenGL_1.py` in this chapter's code section on the CD, also sets the precedent for each PyOpenGL example that follows, so pay attention!

Presenting a Window in PyOpenGL

If you look at the sample code, the first thing you do after giving Python and Pygame access to the PyOpenGL libraries through `import` statements is to declare a couple of variables, like so:

```
rquad = 0.0
xrot = yrot = zrot = 0.0
textures = [0,0]
```

These are variables you'll use in later examples, not for this first simple one, so you can ignore them for now.

After the variables you define how to size the window or PyOpenGL scene. Do this by creating a `windowsize` function. This function will be called to set up the window or scene at least once when the program is first run, and when it is called, it will be given the height and width you want the window to be:

```
def windowresize((width, height)):
    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluPerspective(45, 1.0*width/height, 0.1, 100.0)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
```

The first command in `windowresize` is `glViewport`. This command resets the current view.

The `glMatrixMode(GL_PROJECTION`) line then sets up the projection matrix, which is responsible for adding perspective. `glMatrixMode` is defined by the next two commands, in which the scene is set and the perspective is defined. The command that follows is `glLoadIdentity()`, which resets and restores the projection matrix to its original state.

Objects on the screen that are meant to be far away need to appear smaller in order to create realistic 3D, so the perspective is then defined with `gluPerspective`. In this example, the perspective is calculated by a 45-degree viewing angle based on 1 times (1.0*) `windowsize`'s height and width. 0.1 and 100.0 are the starting and ending points for how deep the screen can go, and how many layers the screen can have.

Finally, you use `glLoadIdentity()` a second time to turn attention to the projection matrix and reset it.

Initializing PyOpenGL

After defining a three-dimensional window, you can then create a function that initializes PyOpenGL. You need to establish what color the screen starts out as, the depth buffer, and whether to use smooth shading, as well as a number of other possible PyOpenGL features. Do this with an `initialize` command:

```
def initialize():
    glShadeModel(GL_SMOOTH)
    glClearColor(0.0, 0.0, 0.0, 0.0)
    glClearDepth(1.0)
    glEnable(GL_DEPTH_TEST)
    glDepthFunc(GL_LEQUAL)
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)
```

In `initialize`, you use `glShadeModel(GL_SMOOTH)` first to ask PyOpenGL to use smooth shading (smooth shading is simply one way of blending colors and lighting when rendering a polygon). Next you use `glClearColor`, which sets the color of the window screen when it is clear.

PyOpenGL takes in four numbers when you declare a color. The first three represent the primary colors red, green, and blue, and the last is the alpha (transparency channel). Each number can range from 0.0 to 1.0; the lower the number, the darker the intensity, the higher the number, the brighter the intensity. The numbers must be in order of Red, Green, Blue, and Alpha. You can create different colors by mixing these primary colors. Black would be (0,0,0,0), white would be (1,1,1,0), and yellow would be (1,1,0,0) . Of course, the last number is the alpha or transparency.

After setting the screen color you set up the depth buffer. The depth buffer keeps track of how many layers deep the screen goes, and you need to have depth in order to have any sort of 3D. The depth buffer actually keeps track of which objects are in front and which are in back, so it knows how to draw the screen in the proper perspective. There are three commands associated with the depth buffer in our `initialize` function: `glClearDepth`, `glEnable`, and `glDepthFunc`.

`glClearDepth` specifies the depth value used when the depth buffer is cleared. The `glEnable` command is used to enable various PyOpenGL capabilities. In this case, it is enabling depth testing, which will allow `initialize` to do depth comparisons and update the depth buffer. `glDepthFunc` specifies the function used to compare each incoming pixel depth value with the depth value present in the depth buffer. LEQUAL is short for Less than or Equal to, and sets `glDepthFunc` to pass the incoming depth value if it is less than or equal to the present value.

`glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST`) is a long command, but it's basically only a way of telling PyOpenGL to please use the best corrective perspective and the highest-quality view when there is room for interpretation.

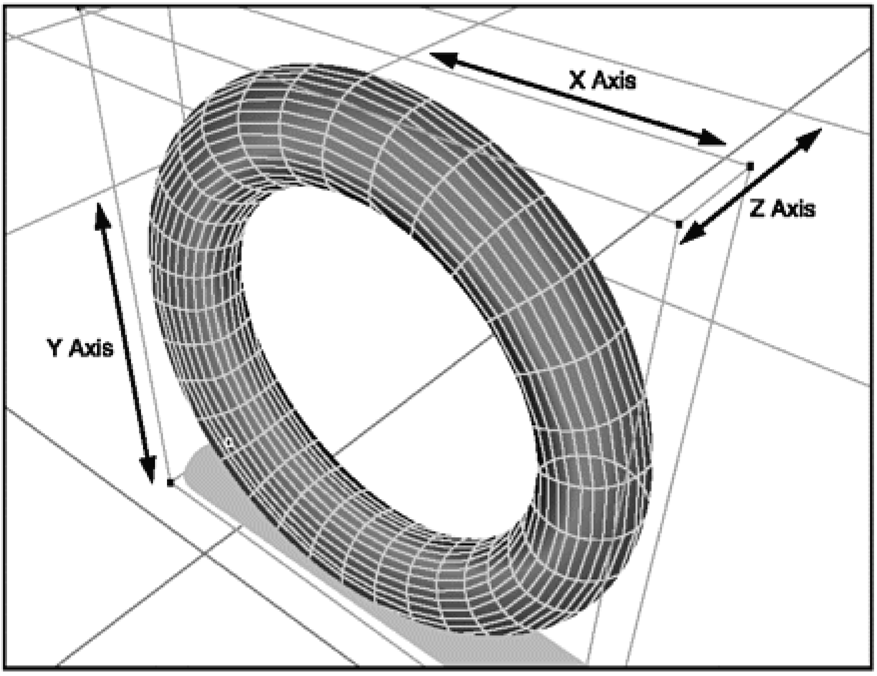
Drawing a Square

Our third function is the code that actually draws the display, so let's call it `drawgraphics()`. This function will actually display everything that goes onto the screen, so it will be doing most of the work in each example.

```
def drawgraphics():
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    glTranslatef(0.0, 0.0, -5.0)
    glBegin(GL_QUADS)
    glVertex3f(-1.0, 1.0, 0)
    glVertex3f(1.0, 1.0, 0)
    glVertex3f(1.0, -1.0, 0)
    glVertex3f(-1.0, -1.0, 0)
    glEnd()
```

First you `glClear` to clear the screen to a color, clear the buffer, and then reset with `glLoadIdentity`. `glLoadIdentity` actually moves you to the center of the screen, which is 0,0 on the x- and y-axis. Left and down are negative numbers, and right and up would be positive numbers; see Figure 4.15.

Figure 4.15. Three-dimensional space labeled by X,Y, and Z



The `glTranslatef()` command produces a translation of the current matrix by multiplying it by the x, y, and z coordinates given to it. This sounds confusing, but all it really does is change the drawing point from the current view to someplace else. In this case, you do not change the `glTranslatef()` x or y coordinates (leaving them at 0,0) but you do give a -5.0 for the z-axis, which basically pushes the matrix back five screen depths. If you didn't push the matrix back, what you drew would be too close to the front of the 3D space for you to see it. Basically, `glTranslatef()` is the command that moves along the x-, y-, and z-axes. For instance, `glTranslatef(1.5, 0.0, and -6.0)` would mean to move left 1.5 units and into the screen depth by 6 units.

NOTE

TIP

When you use `glTranslatef()`, you are not moving coordinated relative to the center of the screen, you are actually moving `glTranslatef()` relative to wherever it currently is. If you left `glTranslatef()` at the top right corner of the screen with the last command, that is where it will still be when you use it later. This means you need always keep track of its current position.

glBegin tells PyOpenGL that you want to start drawing, and (GL_QUADS) tells PyOpenGL that you want to draw a square or four-sided shape of some sort. You use glVertex() to tell PyOpenGL where the four points of your square shape are located on the x-, y-, and z-axes, and glEnd() means you are done drawing and that there are no more points. The first glVertex() number is is the first point of the square (and the x-axis, if you are drawing a polygon). The second number is the y-axis, and the third number is the z.

You have three usable functions; now you just have to set them up in a main loop.

```
def main():
    # Define any variables
    video_flags = OPENGL|DOUBLEBUF
    # Initialize Pygame
    pygame.init()
    pygame.display.set_mode((640,480), video_flags)
    # Call our windowsize and Initialize functions
    windowsize((640,480))
    initialize()
    #set frames to 0 before loop starts
    frames = 0
    # Have pygame keep track of time
    ticks = pygame.time.get_ticks()
    # while loop that draws and looks to quit
    while 1:
        event = pygame.event.poll()
        if event.type == QUIT or (event.type == KEYDOWN and event.key
== K_ESCAPE):
            break
         # Draw our fun graphics
         drawgraphics()
         pygame.display.flip()
         frames = frames+1

if __name__ == '__main__': main()
```

There is actually quite a bit going on here. First, you define video_flags to be OpenGL and double-buffered; these are calls you need to make to Pygame in order to render OpenGL correctly. Then you initialize Pygame with its init() method and set the display to 640x480 with your video flags.

NOTE

Double Buffering

Drawing and redrawing screens and images can be time- and processor-consuming, and game programmers have developed many tricks for increasing the speed it takes to render drawings. One of these tricks is called double buffering, and is very common when animating. Double buffering is so common, in fact, that most modern game and animation libraries have built-in support for flags for using the technique. Can you believe that programmers used to have to create their own buffers by hand? Talk about Dark Ages!

Normally, when an image is redrawn, it is simply redrawn in place on the screen. In double buffering, the image is redrawn ahead of time in a buffer or a hidden area of the screen or memory, and then, when it is time to re-display, the buffer is simply copied to the screen. In reality, a complex animation or sequence may have dozens of unseen layers constantly loading with the graphics that will display seconds later.

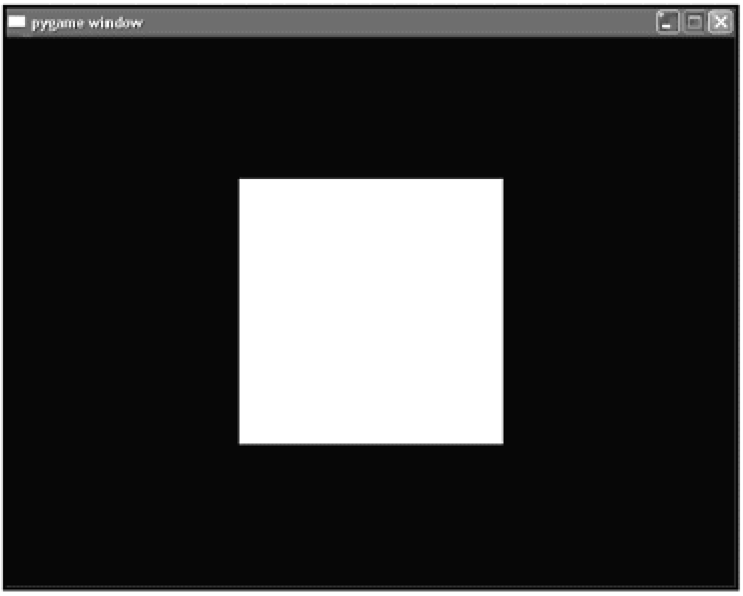
Then you call the `windowsize` function with the same display size (640x480) and the `initialize` function that initializes PyOpenGL. You set up a baseline frame variable (equaling 0) and then you ask Pygame to use `pygame.get.ticks` to keep track of time in milliseconds.

The actual work happens in the `while` loop. First, use Pygame's `event.poll()`function to see, via keyboard input and an `if` statement, whether the user wants to quit. Then call the `draw graphics`function, which draws the square.

`pygame.display.flip()` updates the display each time it is called. `pygame.dipsplay` knows that you are using OpenGL and double buffering because of your earlier video flags, so it updates the entire display by swapping the current view with the new ones it has drawn and stored in memory (this is called a gl buffer swap). Then you update your frames so that you know how many times the `while` loop has looped, and finally you initiate `main` with a standard Python `if` line.

Whew! If you run `OpenGL_1.py` you'll see a white square open in a 640x480-pixel Pygame window, similar to that in Figure 4.16.

Figure 4.16. OpenGL_1.py displays a square rendered in PyOpenGL and displayed within a Pygame window



Setting the Color of an Object

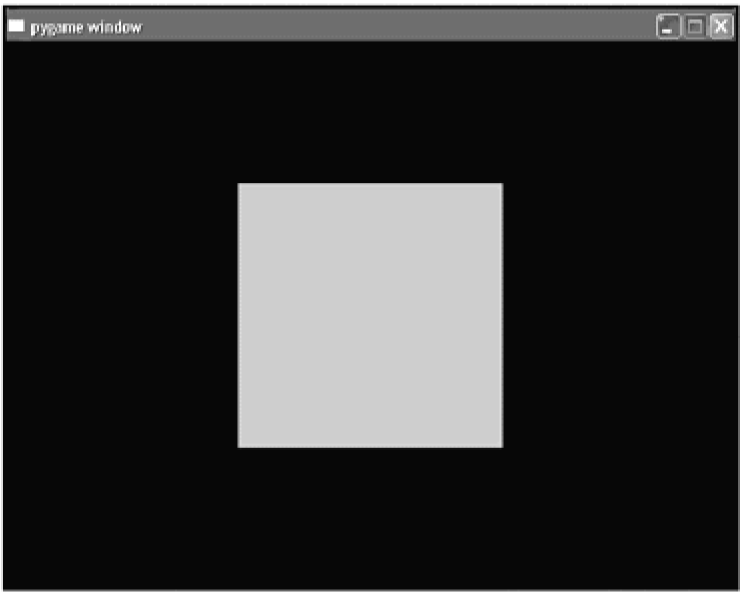
Now that you have a baseline, let's look at what else you can do with PyOpenGL. Let's try giving the square a color. You can use the `glColor3f()` command, which also takes in three commands, one each for red, green, and blue intensity values: `glColor3f(r, g, b,)`. PyOpenGL keeps these standards consistent across commands, so the colors have a range from 0.0 to 1.0 and work exactly the same as if you were setting up the screen background color with `glClearColor3f()`.

Turning on `glColor3f` is like switching to a different-colored pen. When you switch to red, everything you draw after that point is red. Then, if you switch to another color, everything you draw after that is drawn in the new color. To make your square a Python green, you simply need to add the `glColor3f` command in your `drawgraphics()` function before you begin drawing with `glBegin(GL_QUADS)`, like so:

```
def drawgraphics():
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    glTranslatef(0.0, 0.0, -5.0)
   #Adding color to our square
    glColor3f(0.1, 0.9, 0.5)
    glBegin(GL_QUADS)
    glVertex3f(-1.0, 1.0, 0)
    glVertex3f(1.0, 1.0, 0)
    glVertex3f(1.0, -1.0, 0)
    glVertex3f(-1.0, -1.0, 0)
    glEnd()
```

Now when you run this program (labeled `OpenGL_2.py` on the CD), you will see a green square just like that in Figure 4.17. Notice that the polygon fills in the entire surface with the colors you've drawn. This is called smooth coloring.

Figure 4.17. Coloring in a surface with `glColor3f()`



Rotation and Movement

Now that you can color the square, let's try to rotate it. To do so, you need to add a bit to the `drawgraphics` function. First, make use of the `rquad` (`rquad` is short for rotate quad) variable by declaring it a global and then calling the `glRotatef()` function. Use a variable for rotation so that you have fine-grain control over the movement.

`glRotatef(angle, x, y, z)` produces a rotation of a given angle in degrees over a given vertices given in x, y, and z coordinates. The command takes four arguments: `Angle`, `X vector`, `Y vector`, and `Z vector`. `Angle` is a number that represents how much to spin the object. The x, y, and z vectors represent the vector around which the rotation will occur. For instance, (1,0,0), describes a vector that travels in the direction of 1 unit along the x-axis.

The current matrix (remember it's all about `glMatrixMode`) is changed by this rotation. Set up the rotation by adding one line that calls `glRotate()` on your square using the `rquad` variable as the angle and rotating on the x-axis:

```
def drawgraphics():
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    glTranslatef(0.0, 0.0, -5.0)

    # Set up rquad for rotation, only real difference
    global rquad
    glRotatef(rquad, 1.0, 0.0, 0.0)

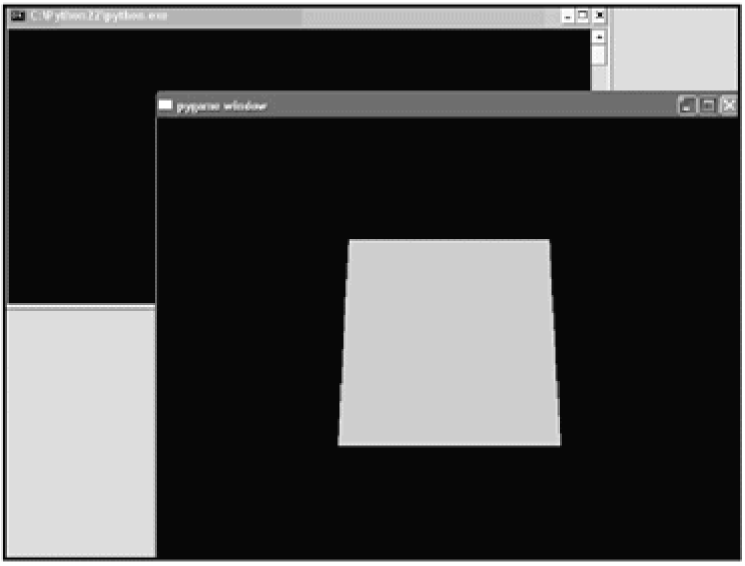
    glColor3f(0.1, 0.9, 0.5)
    glBegin(GL_QUADS)
    glVertex3f(-1.0, 1.0, 0)
    glVertex3f(1.0, 1.0, 0)
    glVertex3f(1.0, -1.0, 0)
    glVertex3f(-1.0, -1.0, 0)
    glEnd()
```

And then at the end of `drawgraphics` you update `rquad` so that the drawing of the square continually rotates:

```
    # And update rquad for movement
    rquad+= 0.1
```

This creates a rotating flat square, as illustrated in Figure 4.18 (the source is on the CD as `OpenGL_3.py`).

Figure 4.18. A flat plane rotates along its x-axis

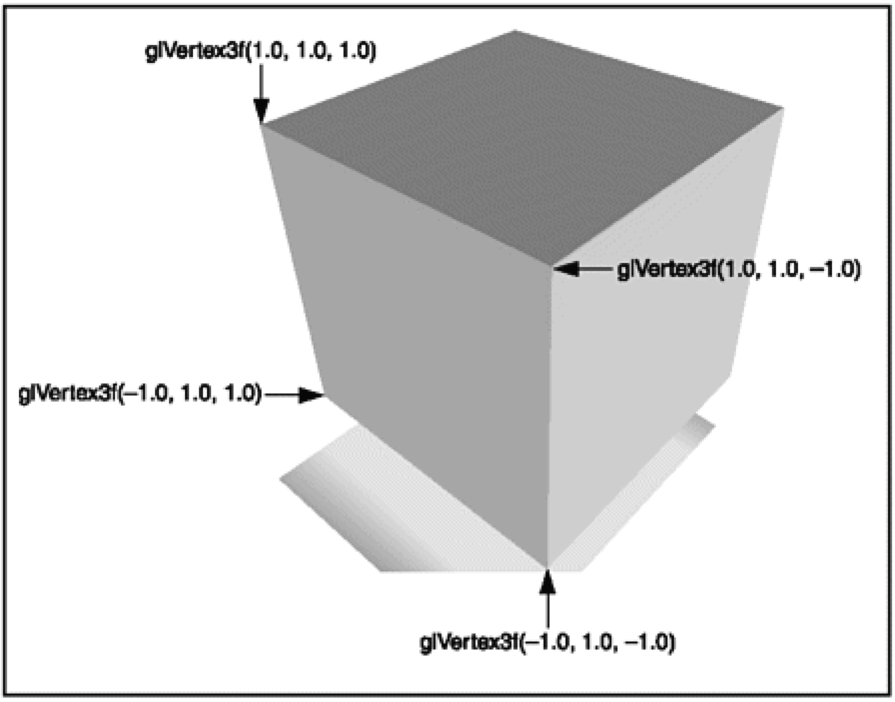


By playing with the `rquad` variable, you can change how many degrees the plane rotates on the x-axis. You can make the plane spin faster or slower, backwards or forwards, by changing the values associated with it.

Moving from Flat to 3D

You have already done most of the work for displaying three dimensions. Let's say you wanted to change your flat plane to a cube. `GL_QUAD` is actually capable of displaying a cube object; you just need to tell it where the other vertices for the other five flat planes should go. This becomes a pixel-plotting problem; it is shown in Figure 4.19.

Figure 4.19. A cube and points for each side are mapped out in 3D space



Once you know where each pixel belongs, you can feed the location to `GL_QUAD`, which fills in each surface for you:

```
# Front Face
glVertex3f( 1.0, 1.0,-1.0)
glVertex3f(-1.0, 1.0,-1.0)
glVertex3f(-1.0, 1.0, 1.0)
glVertex3f( 1.0, 1.0, 1.0)

# Back Face
glVertex3f( 1.0,-1.0, 1.0)
glVertex3f(-1.0,-1.0, 1.0)
glVertex3f(-1.0,-1.0,-1.0)
glVertex3f( 1.0,-1.0,-1.0)

# Top Face
glVertex3f( 1.0, 1.0, 1.0)
glVertex3f(-1.0, 1.0, 1.0)
glVertex3f(-1.0,-1.0, 1.0)
glVertex3f( 1.0,-1.0, 1.0)

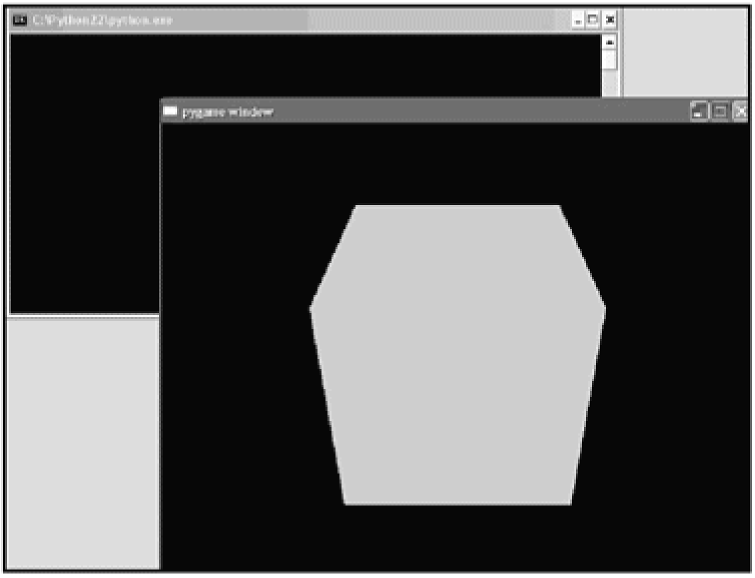
# Bottom Face
glVertex3f( 1.0,-1.0,-1.0)
glVertex3f(-1.0,-1.0,-1.0)
glVertex3f(-1.0, 1.0,-1.0)
glVertex3f( 1.0, 1.0,-1.0)

# Right face
glVertex3f(-1.0, 1.0, 1.0)
glVertex3f(-1.0, 1.0,-1.0)
glVertex3f(-1.0,-1.0,-1.0)
glVertex3f(-1.0,-1.0, 1.0)

# Left Face
glVertex3f( 1.0, 1.0,-1.0)
glVertex3f( 1.0, 1.0, 1.0)
glVertex3f( 1.0,-1.0, 1.0)
glVertex3f( 1.0,-1.0,-1.0)
```

Now the cube has six sides. PyOpenGL automatically draws them in a counter-clockwise order—the first point is top-right, the second point is bottom-right, and so on until completely around the given plane. The rotation is already built-in, and the MatrixMode automatically knows to update each side as it rotates; check out `OpenGL_4.py` on the CD and Figure 4.20.

Figure 4.20. The flat plane becomes a full rotating cube



Let's say you wanted to speed up and twist your rotating cube around a bit more. It's easy to fiddle with MatrixMode, especially since you've thought ahead and included a number of variables with which to do it:

```
# Now we use all of these
# x,y, and z rots are the rotations on each axis
xrot = yrot = zrot = 0.0
```

These variables, `xrot`, `yrot`, and `zrot`, can be used to rotate the cube in a new way on the x-, y-, and x-axes. Do so by adding a few lines to the top of `drawgraphics`:

```
global xrot, yrot, zrot
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    glTranslatef(0.0, 0.0, -5.0)

    global rquad # not used for now
    glRotatef(xrot,1.0,0.0,0.0)
    glRotatef(yrot,0.0,1.0,0.0)
    glRotatef(zrot,0.0,0.0,1.0)
```

And then add a few lines to the end of `drawgraphics`:

```
    # Use XYZ to rotate - speed it up a bit
    xrot = xrot + 0.9
    yrot = yrot + 0.9
    zrot = zrot + 0.9
```

This will cause your cube to rotate quicker and also spin on aother axis.

Adding Textures

In your final PyOpenGL tutorial you'll open and use a local texture image instead of having PyOpenGL simply color the cube; this is illustrated in Figure 4.21. The full code is listed in `OpenGL_5.py` in the Chapter 4 code section on the CD.

Figure 4.21. A textured cube spins around each axis



First you will make use of `import os`. A texture will then have to be loaded from outside of Python, and your program will need to understand how to navigate through different directories and pull files from its native operating system.

You will also finally be using the texture variables you initialized early on:

```
# textures for loading the .bmp image
textures = [0,0]
```

You will be using `textures[]` for loading the .bmp you will be using for texture. The first thing you need is a new function that opens up the .bmp file:

```
# New function to find, load, and use the texture
def loadtextures():
    # Need to find and load the texture
    point_to_file = os.path.join('dtcfe.bmp')
    texture_surface = pygame.image.load(point_to_file)
    texture_buffer = pygame.image.tostring(texture_surface, "RGBX", 1)
```

First, `point_to_file` uses the `os` module's `os.path.join` to point to the .bmp you want to use—in this case it is the dtcfe.bmp file found on the CD with the code samples. The next two commands use Pygame methods to load the .bmp image to a new surface (`texture_surface`) and then copy the image into a larger string buffer (`texture_buffer`). Specifying RGBX tells Pygame that the texture should be 32-bit padded RGB data. This turns the .bmp image into an actual texture.

With Pygame, your textures must be at least 64x64 pixels, and shouldn't be more than 256x256 pixels. Textures need to be sized in height and width to the power of 2 (if the textures are 64x64, 128x128, or 256x256, they do not need to be resized, otherwise they do). These are of course the standard defaults for textures and are changeable, but not without more advanced programming.

Now that Pygame has the texture, you hand it over to OpenGL. First you need to specify that the texture is two-dimensional with `GL_TEXTURE_2D`, and then you need to bind it to a `texture[]` array that will hold any and all textures your program needs:

```
    glBindTexture(GL_TEXTURE_2D, textures[0])
```

`glTextImage2D` is a PyOpenGL command that specifies a two-dimensional texture. You feed it several values, including the texture surface, width, and height (using the `get_width()` and `get_height()` methods). Then you specify that the texture is two-dimensional with `GL_TEXTURE_2D`, explain how the color format is organized with `GL_RGBA`, define the data format used to store the texture data with `GL_UNSIGNED_BYTE`, and finally, you give `glTextImage2D()` the actual data of the texture itself, `texture_buffer`, which you defined with Pygame:

```
    glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA,
texture_surface.get_width(), texture_sur-
face.get_height(), 0,
                  GL_RGBA, GL_UNSIGNED_BYTE, texture_buffer );
```

Whew—that's our longest one-liner yet. The last step in loading a texture is to tell PyOpenGL what filtering to use when the image is stretched or altered on the screen. To do so, use PyOpenGL's built-in `glTexParameterf()`, which simply defines the options to use when texture mapping. The `MIN` and `MAG` filters specify texture magnification, and `GL_NEAREST` asks PyOpenGL to grab the nearest pixel when redrawing the `GL_TEXTURE_2D` image:

```
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
```

Now that you can load the .bmp image and turn it into a texture, you need to make PyOpenGL use the texture on each side of the cube instead of filling in the sides with `glColor3f()`.

Drawing a textured cube is quite a bit different from drawing colored cubes. Most of the gl functions are the same but the `glBindTexture` command we used to load textures sets the texture we want to use, much like `glColor3f()` set the pen to a specific color:

```
glBindTexture(GL_TEXTURE_2D, textures[0])
```

To map the texture correctly into a specific side of the texture, you need to make sure the top-right of the texture is mapped to the top-right of the side; same with the bottom-left. Each corner needs to be mapped using the `glTexCoord2f()`, command like so:

```
    glTexCoord2f(0.0, 0.0); glVertex3f(-1.0, -1.0,  1.0)
```

The `glTexCoord2f` command is designed to map out textures in two dimensions. Once you get the hang of using the command it is as easy to use as `glColor`, there is just an added complexity to each of the cube's mapped points:

```
glBegin(GL_QUADS)

    # Front Face
    glTexCoord2f(0.0, 0.0); glVertex3f(-1.0, -1.0,  1.0)
    glTexCoord2f(1.0, 0.0); glVertex3f( 1.0, -1.0,  1.0)
    glTexCoord2f(1.0, 1.0); glVertex3f( 1.0,  1.0,  1.0)
    glTexCoord2f(0.0, 1.0); glVertex3f(-1.0,  1.0,  1.0)

    # Back Face
    glTexCoord2f(1.0, 0.0); glVertex3f(-1.0, -1.0, -1.0)
    glTexCoord2f(1.0, 1.0); glVertex3f(-1.0,  1.0, -1.0)
    glTexCoord2f(0.0, 1.0); glVertex3f( 1.0,  1.0, -1.0)
    glTexCoord2f(0.0, 0.0); glVertex3f( 1.0, -1.0, -1.0)

    # Top Face
    glTexCoord2f(0.0, 1.0); glVertex3f(-1.0,  1.0, -1.0)
    glTexCoord2f(0.0, 0.0); glVertex3f(-1.0,  1.0,  1.0)
    glTexCoord2f(1.0, 0.0); glVertex3f( 1.0,  1.0,  1.0)
    glTexCoord2f(1.0, 1.0); glVertex3f( 1.0,  1.0, -1.0)

    # Bottom Face
    glTexCoord2f(1.0, 1.0); glVertex3f(-1.0, -1.0, -1.0)
    glTexCoord2f(0.0, 1.0); glVertex3f( 1.0, -1.0, -1.0)
    glTexCoord2f(0.0, 0.0); glVertex3f( 1.0, -1.0,  1.0)
    glTexCoord2f(1.0, 0.0); glVertex3f(-1.0, -1.0,  1.0)

    # Right face
    glTexCoord2f(1.0, 0.0); glVertex3f( 1.0, -1.0, -1.0)
    glTexCoord2f(1.0, 1.0); glVertex3f( 1.0,  1.0, -1.0)
    glTexCoord2f(0.0, 1.0); glVertex3f( 1.0,  1.0,  1.0)
    glTexCoord2f(0.0, 0.0); glVertex3f( 1.0, -1.0,  1.0)

    # Left Face
    glTexCoord2f(0.0, 0.0); glVertex3f(-1.0, -1.0, -1.0)
    glTexCoord2f(1.0, 0.0); glVertex3f(-1.0, -1.0,  1.0)
    glTexCoord2f(1.0, 1.0); glVertex3f(-1.0,  1.0,  1.0)
    glTexCoord2f(0.0, 1.0); glVertex3f(-1.0,  1.0, -1.0)

    glEnd();
```

The result of this code `(OpenGL_5.py)` is illustrated in Figure 4.21.

Sound in Python

Like with graphics, there are a number of available libraries for implementing sound in Python.

- **PythonWare Sound Toolkit.** An (unfortunately) abandoned kit for reading and playing AU, VOC, and WAV files on Windows and Sun OSs. The unfinished tookit is still available from PythonWare at http://www.pythonware.com.

  PythonWare is a copyrighted, but free to use, library.

- **Boodler.** An interesting tool for creating soundscapes which uses Python and is created for UNIX operating systems (although some work on PDAs, Mac, and with Direct X has been done). The project can be found at http://www.eblong.com/zarf/boodler/.

  Boodler combines sound samples into an ongoing stream of sound for background noise.

- **The Snack Toolkit.** The Snack Toolkit was developed by Kare Sjolander for TCL and Python. It is a sound-processing toolkit with a TK interface. It supports MP3 and sound filtering; the idea behind the kit is rapid development. Snack needs both Tkinter and Tcl/Tk to work correctly. It adds the `snack:sound` command, which is used to create and handle sound objects, read audio data from wav files, and play sounds. Snack is accessed using the tkSnack module. You can find information on Snack at http://www.speech.kth.se/snack.
- **The MusicKit Library.** MusicKit is a full, object-oriented library for signal processing and building sound, music, and creating MIDI applications. The kit is based on Music V (From Bell Labs and Max Mathews) and was originally written for NeXT. These are C tools made available to Python using PyObjC or the Objective-C bridge. The DSP tools are only portable to Intel systems or m68k, but the MIDI and sound streaming are available on Windows and Mac platforms (at the time of this writing the project team was still working on a port to Linux). The kit can be found on its own Sourceforge page, along with on-line documentation, code examples, utilities, applications, and musical scores at http://musickit.sourceforge.net/.

Python, of course, comes with a few sound functions built-in. These are included under Multimedia Services and listed in Table 4.27.

Table 4.27. Python Multimedia Audio Services

Module	Use
`audioop`	Manipulates raw audio data.Operates on sound fragments consisting of signed integer samples 8, 16, or 32 bits wide, stored in Python strings
`aifc`	Reads and writes audio files in AIFF or AIFC format (Audio Interchange File Format)
`sunau`	An interface to the Sun AU sound format

Table 4.27. Python Multimedia Audio Services

Module	Use
`wave`	An interface to the WAV sound format. Supports stereo and mono but not compression and decompression
`chunk`	Reads EA IFF chunks
`sndhdr`	Provides utility functions that determine the type of a sound file

Python also possesses a Winsound module that provides access to the basic sound-playing machinery on Windows platforms. Winsound includes a single function from the platform API, `PlaySound`, which takes in a sound parameter argument that can be either a filename, a string (that's a string of audio data) or None.

Winsound's flags are listed in Table 4.28.

Table 4.28. Windsound's Flags

Flag	Purpose
`SND_FILENAME`	The sound parameter is the name of a WAV file
`SND_ALIAS`	The sound parameter should be interpreted as a control panel sound association name
`SND_LOOP`	Play the sound repeatedly
`SND_MEMORY`	The sound parameter to `PlaySound()` is a memory image of a WAV file
`SND_PURGE`	Stop playing a specified sound
`SND_ASYNC`	Allows sounds to play asynchronously
`SND_NODEFAULT`	If the specified sound cannot be found, do not play the default beep
`SND_NOSTOP`	Do not interrupt sounds currently playing
`SND_NOWAIT`	Return immediately if the sound driver is busy

Although loading and playing sounds is covered in this section, audio programming and the science behind sound waves is a complex and in-depth field. If you find audio programming to be your bliss, I suggest checking out a copy of Mason McCuskey's Beginning Game Audio Programming from your local library.

Playing a Sound with Pygame

You can play a sound using Python Pygame with just a few short lines of code. First do the typical pygame import and the os module import so that you can find files on the native operating system:

```
# Import necessary modules
import os, pygame
from pygame.locals import *
```

After importing the needed libraries, you initialize pygame:

```
pygame.init()
```

Pygame 's cross-platform music tools for sound effects and music are built through the mixer module, so you use `pygame.mixer` to load the sound, and the built-in `play()` method to play it:

```
sound1 = pygame.mixer.Sound('JUNGLE.wav')
sound1.play()
```

That's it. To get this code to run on its own (as the `Play_Sound.py` sample in the Chapter 4 code section on the CD does), you also need to add a loop that keeps the program running so that the sound has time to be loaded and played:

```
while 1: pass
```

Viola! Instant sound with only six small lines of code! Not bad at all. Of course, a real game will need a `sound` function that's a bit more versatile.

Building a load_sound Function

A Pygame `load_sound` function would look very similar to the `load_image` function you created at the beginning of this chapter. You start by defining the function, which takes in the name of the sound file:

```
def load_sound(name):
```

The `load_sound` code should check to see if `pygame.mixer` (the Pygame module that loads up sounds) is installed. If `pygame.mixer` isn't available, Pygame will not be able to load the sound. Pygame has a built-in feature called Nonesound, which, if used, will send a blank sound object if the file cannot be found, so your function will not crash while trying to load a non-existent sound.

```
if not pygame.mixer:
        return NoneSound()
```

Next, as with `load_image`, you build the complete path to the object with the os module:

```
fullname=os.path.join('data', name)
```

Then use a `try/except` clause and return the `sound` object:

```
try:
        sound=pygame.mixer.Sound(fullname)
except pygame.error, message:
        print 'Cannot load sound:', wav
                raise SystemExit, message
        return sound
```

The full snip can be found as `Load_Sound.py` on the CD:

```
def load_sound(name):
        class NoneSound:
                def play(self): pass
        if not pygame.mixer:
                return NoneSound()
        fullname=os.path.join('data', name)
        try:
                sound=pygame.mixer.Sound(fullname)
        except pygame.error, message:
                print 'Cannot load sound:', wav
                raise SystemExit, message
        return sound
```

Networking in Python

For Python to send or receive information between two computers, it needs both of those computers to understand a common address. This address consists of two things: an Internet address (or IP address) and a port number.

IP addresses are 32-bit numbers represented by four decimals and separated by dots (for example: 10.124.220.13). These numbers range from 0 to 255. Each IP address for each network card or connector in a network must be unique.

A port is an entry point into an application or service that resides on the computer. Ports are numbers represented by 16-bit integers, ranging from 0 to 65-535. Certain ports on any

NOTE

The OSI Model

Systems of networking are defined by the OSI/ISO (Open Systems Interconnection/International Standards Organization) model. The OSI model is made up of seven layers. Most of today's networking protocols (like TCP/IP and UDP) span a few of these layers.

1. Physical Layer.

   Defines the information needed to transport data over physical components (cables).

2. Data Link Layer.

   Defines how data is passed to and from the physical components.

3. Network Layer.

   Organizes the network by assigning addresses to each network element (IP).

4. Transport Layer.

   Packs data and ensures transfer on the network (TCP, UDP).

5. Session Layer.

   Handles each individual session or connection made.

6. Presentation Layer.

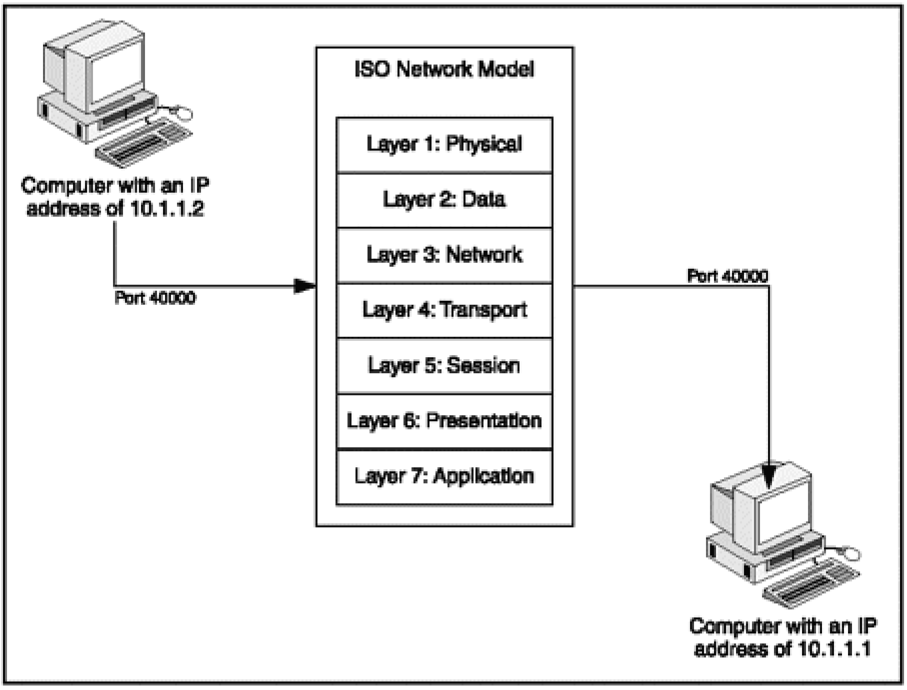
   Used to handle problems with different formats and platforms.

7. Application Layer.

The actual application—the FTP client, HTTP browser, e-mail handlers, and so on, that run on the network.

given machine are responsible for connections to certain services and applications (for instance, port 80 is reserved for HTTP or Web page requests). Any number less than 1,024 is considered privileged, or reserved, and on most computer systems you will need to be an administrator of some sort to run an application on them. An example of this process is outlined in Figure 4.22 along with the OSI network model (see sidebar).

Figure 4.22. Sample communication between two computer stations



Python uses a construct called a socket to send and receive data between addresses. Sockets were originally introduced by UNIX BSD way back in the early 80s and are used today to provide network-application connections. Basically, each end of a network application needs to have a `socket` object of some type established on an address in order to send and receive data or communicate. Establishing a socket on an address is called binding.

Python has a `socket()` module to create object-based socket-style connections, and `socket()`

Table 4.30. `socket()` Methods

Method	Purpose
`accept()`	Accepts a new connection and returns two values: a new `socket` object to be used to transfer data and the address of the socket that this object is talking to

Table 4.30. `socket()` Methods

Method	Purpose
`bind()`	Binds the socket to a port address
`close()`	Closes the socket
`connect()`	Connects to another socket
`getpeername()`	Returns an IP address and the port to which the socket is connected
`getsocketname()`	Returns an IP address and the port of its own socket
`listen()`	Starts listening on a given port, waiting for other sockets to connect
`makefile()`	Creates a file object that you can use `read()` and `write()` on
`recvfrom()`	Returns the data string received from the socket and the IP address that has originated from the socket
`send()`	Sends the date string to the socket
`sendto()`	Sends the data string to the socket hosted by `hostame` at the provided port
`setblockingflag()`	Blocks all read/write operations
`shutdown()`	Shuts down the client sockets or the server sockets or both

can be used to create both sides of a connection (which are usually referred to as the client- side and server-side). The `socket()` module implements a number of functions, as listed in Table 4.29.

Table 4.29. `socket()` Functions

Function	Purpose
`socket()`	Creates and returns a new `socket` object
`gethostname()`	Returns the hostname of the local machine
`gethostbyname()`	Converts hostname to an IP address
`gethostbyaddr()`	Returns a tuple containing the hostname, hostname alias list, and hostname IP list
`getprotobyname()`	Returns a constant value equivalent to the protocol name
`getservbyname()`	Returns the port number associated to the service and protocol pair

Once created, each `socket` object has access to a number of methods, as listed in Table 4.30.

NOTE

`socket.ssl()` can be used to set up a secure SSL connection. The secure connection uses OpenSSL, which is also supported in the `socket` module.

Let's get Python to create a network connection—in this case, a TCP connection (see the upcoming sidebar for more information on TCP and UDP). In order to set up the server side of the connection, Python needs to take the following steps:

1. Create a socket.

2. Bind the created socket to an available port.

3. Start listening on that port.

4. Check the port periodically for new connections coming in.

5. When a connection comes in (from the client side), the server processes the request and sends it back to the client.

Taken one at a time, these steps are fairly straightforward to implement. To create a socket, you first import the `socket` module and then create an instance of a socket; this requires a call to the socket constructor. The code looks like this:

```
# Import the socket() module
import socket
# Call the socket constructor
created_socket=socket.socket(family, type)
```

Typically, the family designated in the socket constructor is set as `AF_INET`, which is an Internet-type socket, or a socket that communicates between different machines. You may also run into the `AF_UNIX` family, which is used for a UNIX-type socket and is normally used when sockets communicate with each other on the same machine.

For a type designation you would see `SOCK_STREAM` for a stream or TCP connection or `SOCK_DGRAM` for a datagram or UDP connection. If you wanted an Internet TCP connection, the socket constructor would look like this:

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

After creating a socket, you need to bind the socket to a port. To do so, you use the `bind()` method:

```
socket.bind(address)
```

The `socket` is of course replaced with your socket instance, and the address is a two-part tuple in the form of (`host, port`). If you wanted to bind `server_socket` to host 10.100.100.201 and port 9000, do this:

```
server_socket.bind("10.100.100.201", 9000)
```

Step 3 is to tell the server to start listening on the port, waiting for any connections. For this step you use the `listen()` method, like so:

```
socket.listen(backlog)
```

`backlog` lists the maximum number of clients that can request connections from the server. In this example, you will set `server_socket` with a maximum of 10 connections:

```
server_socket.listen(10)
```

Now you need to set up a loop that waits for the client to request a connection. The loop needs to run an `accept` method to receive the client requests:

```
connection, address = socket.accept()
```

Finally, you set up communications for the server and client using the `send()` and `recv()` methods. All of this inside of a `while` loop in Python looks like the following:

```
while 1:
        data_sent = "data to send client"
        client_socket, client_address = server_socket.accept()
        print "Connection established with", client_address
        client_socket.send(data_sent)
        client_socket.close()
```

The `data_sent` variable sets the data that will be sent to the client. Then the socket `accept()` method grabs the client address to print on the following line. The contents of `data_sent` are then sent to the client, and the connection is closed with the `close()` method.

The connection on the client side is even easier to code. There are only three things that need to be done:

NOTE

TCP versus UDP

TCP/IP is a connection-oriented form of networking. It was originally developed by the US Department of Defense as a form of communication with built-in redundancy. Layer 3 of the OSI model (the Network Layer) is provided by the Internet Protocol (IP), which provides the basic mechanism for routing packets back and forth on the Internet.

TCP is short for Transmission Control Protocol. It is the main form of communication over the Internet (working on OSI's Layer 4). IP needs TCP because on Level 3, IP

doesn't understand the relationships between the packets it sends, and it doesn't perform any re-transmission. TCP handles the reliability by double-checking the packets' arrival and controling sequencing of packets by keeping track of when each one arrives. With TCP and IP, you can have two-way connections between machines over the physical OSI layers, and, thus, all the cable, wires, phone lines, satellites, and wireless stations that make up the Internet.

UDP is a different form of protocol that provides transport on OSI's Level 4 instead of TCP. UDP is faster because it doesn't track packets sent and it doesn't bother acknowledging their arrival. This, of course, is also less reliable. TCP guarantees delivery and the order of delivery, but UDP doesn't guarantee either, and since it doesn't have to waste time to double check, it can send packets to a destination more quickly.

1. First, create a socket.

2. Open a connection to the server socket via the address (the address being the host's IP and the port number it is listening on).

3. If any data comes through the connection, process it and close the connection.

Step 1 looks fairly identical to the server-side steps:

```
import socket
client_socket = socket.socket(socket.AD_INET, SOCK_STREAM)
```

After the socket is created, Step 2 involves connecting via the server address; this is accomplished through the `connect()` method:

```
client_socket.connect("server_hostname", 9000)
```

Finally, any data received is processed via the `recv()` method (capped at 512 bytes in this example), printed, and then the client connection is closed via the `close()` method:

```
data_received = client_socket.recv(512)
client_socket.close()
print "Received from host", data
```

Let's try the sample again, only this time initiate a UDP connection instead of a TCP connection. With UDP, the server still creates a socket and binds with the address and then begins listening. But at that point, the server's obligations stop, and the rest is handled by the client.

To start, when initializing the socket you must specify `SOCK_DGRAM` instead of `SOCK_STREAM`:

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

And, in this case, the `while` loop action is shortened up to only receive the information from the client (with a maximum number of bytes again) and display it:

```
while 1:
        data_sent, address = server_socket.recvfrom(512)
        print address[0], "server sent: ", data_sent
```

It is the client, in this example, that does the rest of the work, Again, you need to specify that the socket is a UDP-type socket:

```
client_socket = socket.socket(socket.AD_INET, SOCK_DGRAM)
```

Then you specify the data to send, make the connection, send the data, and close the connection:

```
data_sent = my_input("Data to send")
client_socket.sendto(data_sent, ("server_hostname", 9000))
client_socket.close()
```

How about an actual example? How about setting up a socket client and a socket server and send text data between them? (This code can be found in the Chapter 4 source files, labeled `UDP_Server.py` and `UDP_Client.py`, on the CD.) For the server, start by importing `socket` and then designate a host and a port as variables:

```
# UDP_Server.py
import socket

My_Host = "127.0.0.1"
My_Port = 5555
```

You are using the standard localhost address 127.0.0.1 because, by doing so, you can then test the server and client on the same machine. Next, establish a UDP socket instance as before, and bind the socket to `My_Host` and `My_Port`:

```
# Create the socket instance
My_Socket = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )

# Bind the socket to host and port
My_Socket.bind( ( My_Host, My_Port ) )
```

And finally, add a `while` loop that receives the packet from the client:

```
while 1:

    Received_Packet, address = My_Socket.recvfrom( 1024 )
    print "Packet received:"
    print "From host:", address[ 0 ]
    print "Host port:", address[ 1 ]
    print "Containing:"
```

```
    print "\n" + Received_Packet
    # Send data back to client
    print "\ndata to client...",
    My_Socket.sendto( Received_Packet, address )
    print "Packet sent\n"
```

This time, you take advantage of the information that comes through the connection, print the data, and then send data back to the client. Afterwards, you close the socket connection.

Now for the client: Again you need to import the socket, set up the variables, and create an instance of the socket:

```
# UDP_Client.py

# Import socket and set up variables
import socket

My_Host = "127.0.0.1"
My_Port = 5555

# Create the socket instance
My_Socket = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
```

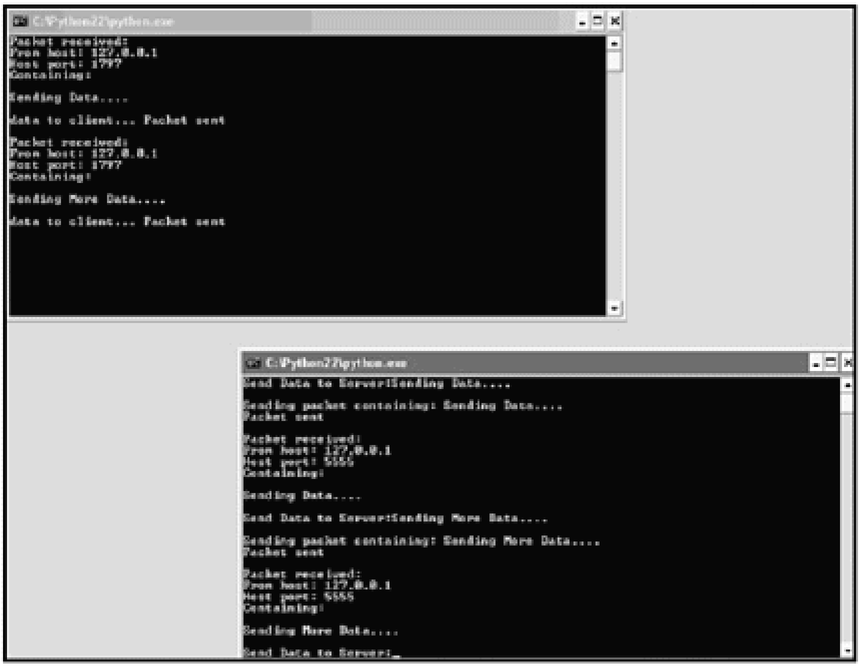
Now you handle the sending of the data back and forth in a `while` loop:

```
# while loop that handles the sending of the packets
while 1:
   # Send the data packet to the server
   My_Packet = raw_input( "Send Data to Server:" )
   print "\nSending packet containing:", My_Packet
   My_Socket.sendto( My_Packet, ( My_Host, My_Port ) )
   print "Packet sent\n"

   # Receive information back from the server
   My_Packet, address = My_Socket.recvfrom( 1024 )
   print "Packet received:"
   print "From host:", address[ 0 ]
   print "Host port:", address[ 1 ]
   print "Containing:"
   print "\n" + My_Packet + "\n"
```

Data is received through Python's useful `raw_input` and sent to the server socket. The `while` loop stays open to receive information that it is expecting from the server, and then prints out the information. When you run the client and server, you are able to send a custom message back and forth; it looks something like Figure 4.23.

Figure 4.23. UDP client server connection using the `socket()` module


C:\Python22\python.exe
Packet received:
From host: 127.0.0.1
Host port: 1797
Containing:
Sending Data....
data to client... Packet sent
Packet received:
From host: 127.0.0.1
Host port: 1797
Containing:
Sending More Data....
data to client... Packet sent
C:\Python22\python.exe
Send Data to Server:Sending Data....
Sending packet containing: Sending Data....
Packet sent
Packet received:
From host: 127.0.0.1
Host port: 5555
Containing:
Sending Data....
Send Data to Server:Sending More Data....
Sending packet containing: Sending More Data....
Packet sent
Packet received:
From host: 127.0.0.1
Host port: 5555
Containing:
Sending More Data....
Send Data to Server:_

Putting It All Together

In this section you'll take a bit from each previous part in the chapter to create a sample game. This sample is called Snowboard!.py and can be found along with its data files in this chapters section on the CD.

Snowboard! has a structure similar to the `Monkey_Toss.py` sample from earlier, and you'll follow the same general steps during creation:

1. Import the necessary libraries.

2. Define any necessary functions, the only one in this case being a `Display_Message` function for displaying splash text on the screen.

3. Define any game object classes, in this case `SimpleSprite`, `Player`, `Obstacle`, and `FinishLine`.

4. Create a `main()` function and set up Pygame.

5. Draw and update the necessary graphics utilizing groups and sprites within a `while()` loop.

Are you ready? Then break!

Import the Libraries

And the libraries are:

```
import os
import sys
import random
import pygame
from pygame.locals import *
```

'Nuff said.

Define the Functions

You want to set up a function that will display text in the game window. Let's call this function `Display_Message`, and use it to display a "You Win!" or a "Game Over!" message at the game's conclusion. The function will take three parameters: the actual message, the game screen, and the game background. You'll use `pygame.font.Font` to define the type of font to use, `font.render` to render the message in white (RGB values 1,1,1), and use `get_rect().cen terx` and `centery` to ensure the text placement is in the center of the window.

```
# Generic function to place a message on screen
def Display_Message( message, screen, background ):
   font = pygame.font.Font( None, 48 )
   text = font.render( message, 1, ( 1, 1, 1 ) )
```

```
    textPosition = text.get_rect()
    textPosition.centerx = background.get_rect().centerx
    textPosition.centery = background.get_rect().centery
    return screen.blit( text, textPosition )
```

As you see, our `Display_Message` function looks remarkably similar to the `font.render` example earlier in the chapter.

Define the Classes

Snowboard! will have four classes: `SimpleSprite`, which will be a base class for all the other classes, and a `Player`, `Obstacle`, and `FinishLine` class:

```
class SimpleSprite:
class Player( SimpleSprite ):
class Obstacle( SimpleSprite ):
class FinishLine( SimpleSprite ):
```

The `SimpleSprite` is the basis for all the others, and defines base methods for placing the sprite on the screen using `blit()` and then covering the sprite with the background to make it disappear. The default `_init_` method can take a loaded image and set itself up as a `rect()`:

```
# Base sprite class for all moving pieces
class SimpleSprite:

    def __init__( self, image ):
        # Can load an image, sets up w-In a rect()
        self.image = image
        self.rectangle = image.get_rect()

    def place( self, screen ):
        #Places the sprite on the given screen
        return screen.blit( self.image, self.rectangle )

    def remove( self, screen, background ):
        #Place under background to remove
        return screen.blit( background, self.rectangle,
            self.rectangle )
```

The `FinishLine` is a sprite that represents a movable line on the game board. The snowboarder must travel a number of screen lengths before reaching the finish, dodging obstacles on his way.

You only need an `_init_` method and a `move` method for `FinishLine` to initialize it and then move it where you have established the end game to be:

```
# Finish line - movable for game difficulty
class FinishLine( SimpleSprite ):

# Initialize and center
    def __init__( self, image, centerX = 0, centerY = 0 ):
```

```
      SimpleSprite.__init__( self, image )
      self.rectangle.centerx = centerX
      self.rectangle.centery = centerY

#Finish line can move up and down depending upon game difficulty
   def move( self, xIncrement, yIncrement ):
      self.rectangle.centerx -= xIncrement
      self.rectangle.centery -= yIncrement
```

The `Obstacle` sprite will be used to load up tree images, which the snowboarder will have to avoid, to place on the screen. Notice how the `move()` method is used:

```
# Class definition for the trees to avoid
class Obstacle( SimpleSprite ):
# Initiate an object of the class
   def __init__( self, image, centerX = 0, centerY = 0 ):
      # Initiate with a loaded image and set as a rectangle
      SimpleSprite.__init__( self, image )
      self.positiveRectangle = self.rectangle
      # move obstacle to a specified location
      self.positiveRectangle.centerx = centerX
      self.positiveRectangle.centery = centerY
      # display that the object has moved position
      self.rectangle = self.positiveRectangle.move( -60, -60 )
```

The movement of these sprites will be dependent upon the player's actions, and will require a complicated `move` method:

```
   def move( self, xIncrement, yIncrement ):
      #Move trees up as the player moves down the slope
      self.positiveRectangle.centerx -= xIncrement
      self.positiveRectangle.centery -= yIncrement
      # Change position for the next sprite update
      if self.positiveRectangle.centery < 25:
         self.positiveRectangle[ 0 ] += \
            random.randrange( -640, 640 )
      # Keep the rectangle values from overflowing
      self.positiveRectangle[ 0 ] %= 760
      self.positiveRectangle[ 1 ] %= 600
      # Display that the object has moved In position
      self.rectangle = self.positiveRectangle.move( -60, -60 )
```

You will also need to check, using a `Collision_Watch` method, for sprite collisions with the snowboarder. The rectangular box that you use to detect the collisions is actually a bit smaller than the graphics:

```
   def Collision_Watch( self ):
      #Make the collision box smaller than graphic
      return self.rectangle.inflate( -20, -20 )
```

Finally, you need to define the `Player` class, which is the class that will actually control the snowboarder. This class must be able to accomplish several things. First, the

snowboarder's four graphics—default, going left, going right, and crashed—all need methods, and a method must also exist to load each graphic when it is needed.

The `Player` class speed should be controllable, which means you need three methods—one to determine if the `Player` class is moving at all, one for speeding up, and one for slowing down.

The `Player` class also needs to watch for collisions with `Obstacle` classes, and remember how far it has traveled so it can know when it passes `FinishLine`. Altogether, this works out to some ten methods:

```
class Player( SimpleSprite ):
   def __init__( self, images, crashImage, centerX = 0, centerY = 0 ):
   def Load_Image( self ):
   def Move_Left( self ):
   def Move_Right( self ):
   def Decrease_Speed( self ):
   def Increase_Speed( self ):
   def Collision( self ):
   def Collision_Watch( self ):
   def Are_We_Moving( self ):
   def Distance_Moved( self ):
```

We start with the `_init_` method that establishes the loading graphic and the initial state of the `Player`:

```
   def __init__( self, images, crashImage,
      centerX = 0, centerY = 0 ):
      # Initial image and player state
      self.movingImages = images
      self.crashImage = crashImage
      # Initial Positioning - top and center
      self.centerX = centerX
      self.centerY = centerY
      # Starts with the Player graphic facing down
      self.playerPosition = 1
      # Start with 0 speed - not moving
      self.speed = 0
      self.Load_Image()
```

You use yet another version of `Load_Image` to pull each version of the snowboarder graphic when needed:

```
# Load the correct image
   def Load_Image( self ):
      # If the player has crashed - special
      if self.playerPosition == -1:
         image = self.crashImage
      else:
         # All other cases the self.playerPosition determines which
graphic to use
         image = self.movingImages[ self.playerPosition ]
      # Notice that the SimpleSprite Is re-Initialized
      SimpleSprite.__init__( self, image )
```

```
      self.rectangle.centerx = self.centerX
      self.rectangle.centery = self.centerY
```

Now tackle movement. The following simply double-check that `Player` class hasn't crashed into something, and then change the player's position:

```
#Player Is Moving left
   def Move_Left( self ):
       # Check for crashing, If so drop speed
       if self.playerPosition == -1:
          self.speed = 1
          self.playerPosition = 0
       # Otherwise start moving left
       elif self.playerPosition > 0:
          self.playerPosition -= 1
       self.Load_Image()

#Player Is Moving Right
def Move_Right( self ):
       #Check for crashing
       if self.playerPosition == -1:
          self.speed = 1
          self.playerPosition = 2
       # Otherwise start moving right
       elif self.playerPosition < ( len( self.movingImages ) - 1 ):
          self.playerPosition += 1
       self.Load_Image()
```

When moving down the hill, the `Player` class will have variable speeds. First use the `Are_We_Moving` method to determine if the `Player` class is moving at all:

```
# Is Player moving or does speed = 0
def Are_We_Moving( self ):
       if self.speed == 0:
          return 0
       else:
          return 1
```

Then we define, increase, and decrease speed, which basically alters from 1 to 10 variables that the game code will use to increase or decrease the `Obstacle` movement rates:

```
# Subtract 1 from speed
def Decrease_Speed( self ):
       if self.speed > 0:
          self.speed -= 1
# Add 1 to speed up to 10,
# Double check to see If we crash
   def Increase_Speed( self ):
      if self.speed < 10:
         self.speed += 1
      # player crashed
      if self.playerPosition == -1:
         self.playerPosition = 1
         self.Load_Image()
```

Next, you need to keep track of the distance the `Player` class has moved. You do this with two variables, `xIncrement` and `yIncrement`. These start at 0 and then increase as the `Player` class moves down the virtual hill. Additionally, if `Player` is facing straight down, she travels a little bit faster than when she is traversing the hill. The distance is also modified by `self.speed`:

```
def Distance_Moved( self ):
      xIncrement, yIncrement = 0, 0
      if self.isMoving():
          # Are we facing straight down, then faster
         if self.playerPosition == 1:
            xIncrement = 0
            yIncrement = 2 * self.speed
         else:
            xIncrement = ( self.playerPosition - 1 ) * self.speed
            yIncrement = self.speed
      return xIncrement, yIncrement
```

Finally, set up collisions. This includes the same sort of `Collision_Watch` you saw earlier with `Obstacle`, and also a `Collsion` method that can change the `Player`classes' graphic if necessary:

```
def Collision_Watch( self ):
      #Slightly smaller box
      return self.rectangle.inflate( -20, -20 )
# Change graphic If necessary
def Collision( self ):
      #Change graphic to player crashed
      self.speed = 0
      self.playerPosition = -1
      self.Load_Image()
```

Create main() and Set Up Pygame

The `main()` function is where all of the fun happens. The game needs a number of variables defined, some of which change constantly and others that never change at all (called constants). The first trick is to get all of these straight.

```
def main():

   #First set Constants (all capitalized by convention)
   # Time to wait between frames
   WAIT_TIME = 20
   # Set the course to be 25 screens long at 480 pixels per screen
   COURSE_DEPTH = 25 * 480
   # Seeds the number of trees on the screen
   NUMBER_TREES = 5
   # Secondly set Variables
   # vertical distance traveled
   distanceTraveled = 0
   # time to generate next frame
   nextTime = 0
```

```
    # The course has not been completed
   courseOver = 0
# Randomly generated obstacle sprites
   allTrees = []
# All screen position sprites that have changed and are now "dirty"
   dirtyRectangles = []
# current time clock
   timePack = None
# Total time to finish course
   timeLeft = 60
```

There are a number of images and sounds you will be using (located in the Data folder under Chapter 4's code listing on the CD), so we need to tell Python where they are exactly and what you will call them:

```
    # The paths to the sounds
   collisionFile = os.path.join( "data", "THUMP.wav" )
   chimeFile = os.path.join( "data", "MMMMM1.wav" )
   startFile = os.path.join( "data", "THX.wav" )
   applauseFile = os.path.join( "data", "WOW2.wav" )
   gameOverFile = os.path.join( "data", "BUZZER.wav" )

   # The paths to the Images
   # Place all snowbaord files Into girlFiles
   girlFiles = []
   girlFiles.append( os.path.join( "data", "surferLeft.gif" ) )
   girlFiles.append( os.path.join( "data", "surfer.gif" ) )
   girlFiles.append( os.path.join( "data", "surferRight.gif" ) )
   girlCrashFile = os.path.join( "data", "surferCrashed.gif" )
   treeFile = os.path.join( "data", "tree.gif" )
   timePackFile = os.path.join( "data", "time.gif" )
   game_background = os.path.join("data", "background2.png")
```

Now, to initialize Pygame, set the game surface to be 640x480 pixels, make the box caption "Snowboard!", and make the mouse invisible, as the game code doesn't use it:

```
   # initializing pygame
   pygame.init()
   screen = pygame.display.set_mode( ( 640, 480 ) )
   pygame.display.set_caption( "Snowboard!" )
   # Make mouse.set_visable = false/0
   pygame.mouse.set_visible( 0 )
```

Now that Pygame has been initialized and you have a window to play in, set the background to the nice snowy-hill-looking background2.png image:

```
   # Grab and convert the background image
   background = pygame.image.load( game_background ).convert()
   # blit the background onto screen and update the entire display
   screen.blit( background, ( 0, 0 ) )
   pygame.display.update()
```

Now you need to use Pygame to load the sounds and images to which you have established the paths:

```
# First load up the sounds using mixer
collisionSound = pygame.mixer.Sound( collisionFile )
chimeSound = pygame.mixer.Sound( chimeFile )
startSound = pygame.mixer.Sound( startFile )
applauseSound = pygame.mixer.Sound( applauseFile )
gameOverSound = pygame.mixer.Sound( gameOverFile )

# Next we load the images, convert to pixel format
# and use colorkey for transparency
loadedImages = []
# Load all the snowboard files which are In girlFiles
# Then append them Into LoadedImages[]
for file in girlFiles:
   surface = pygame.image.load( file ).convert()
   surface.set_colorkey( surface.get_at( ( 0, 0 ) ) )
   loadedImages.append( surface )
# load the crashed surfer image
girlCrashImage = pygame.image.load( girlCrashFile ).convert()
girlCrashImage.set_colorkey( girlCrashImage.get_at( ( 0, 0 ) ) )
# load the tree image
treeImage = pygame.image.load( treeFile ).convert()
treeImage.set_colorkey( treeImage.get_at( ( 0, 0 ) ) )
# load the timePack image
timePackImage = pygame.image.load( timePackFile ).convert()
timePackImage.set_colorkey( surface.get_at( ( 0, 0 ) ) )
```

There are three last things you need to do before jumping into the `while()` game loop. The first is initialize the `Player` snowboarder. Secondly, set up all the `Obstacle` trees on the course. Finally, play the start up THX sound, just for effect:

```
 # initialize the girl-snowboarder
centerX = screen.get_width() / 2
# Create and Instance of Player called theGirl
# Use the crashimage, center horizontally and 25 pixels from the
top
theGirl = Player( loadedImages, girlCrashImage, centerX, 25 )
# place tree Objects in randomly generated spots
for i in range( NUMBER_TREES ):
   allTrees.append( Obstacle( treeImage,
      random.randrange( 0, 760 ), random.randrange( 0, 600 ) ) )
# Play start - up sound for effect
startSound.play()
pygame.time.set_timer( USEREVENT, 1000 )
```

Drawing and Updating within the while Loop

Now you need to set up the `while` loop that updates all the sprites, keeps track of time, and renders everything. The `while` loop will be set to run until the course is over:

```
while not courseOver:
```

Then there are a few things you need to do with timing to make sure the game flows smoothly:

```
currentTime = pygame.time.get_ticks()
# Wait In case we are moving too fast
if currentTime < nextTime:
   pygame.time.delay( nextTime - currentTime )
 # Update the time
nextTime = currentTime + WAIT_TIME
```

Then check for sprites that are "dirty" (that have changed and need to be updated). We remove any sprites that need to be removed and check to see whether a `timePack` should to be drawn (a timePack will increase the time left before the loop is exited, giving the player more time to reach the finish line):

```
# remove objects from screen that should be removed
dirtyRectangles.append( theGirl.remove( screen,
   background ) )
# Check all the trees
for tree in allTrees:
   dirtyRectangles.append( tree.remove( screen,
      background ) )
 # Check timepack
if timePack is not None:
   dirtyRectangles.append( timePack.remove( screen,
      background ) )
```

Now throw in the event code that listens for a player hitting the keyboard. Use Pygame's built in `poll()` method to fill the event queue. The player's commands directly affect the `Player` instance (`theGirl`) by calling the appropriate methods:

```
# get next event from event queue using poll() method
event = pygame.event.poll()
# if player quits program or presses the escape key
if event.type == QUIT or \
   ( event.type == KEYDOWN and event.key == K_ESCAPE ):
   sys.exit()
# if the up arrow key was pressed, slow down!
elif event.type == KEYDOWN and event.key == K_UP:
   theGirl.Decrease_Speed()
# if down arrow key was pressed, speed up!
elif event.type == KEYDOWN and event.key == K_DOWN:
   theGirl.Increase_Speed()
# if right arrow key was pressed, move player right
elif event.type == KEYDOWN and event.key == K_RIGHT:
   theGirl.Move_Right()
# if left arrow key was pressed, move player left
elif event.type == KEYDOWN and event.key == K_LEFT:
   theGirl.Move_Left()
# Update the time that the player has left
elif event.type == USEREVENT:
   timeLeft -= 1
```

Use `random` to randomly create `timePacks` on the screen as the player travels down the mountain:

```
# 1 in 100 odds of creating new timePack
      if timePack is None and not random.randrange( 100 ):
         timePack = FinishLine( timePackImage,
            random.randrange( 0, 640 ), 480 )
```

Now, as the `theGirl` class instance moves down the mountain, you need to make sure the sprites that handle the trees and the `timePack` are updated and redrawn. This only happens if `Are_We_Moving` is true:

```
# update obstacles and timePack positions if the player Is moving
      # First check Are_We_Moving
      if theGirl.Are_We_Moving():
         # Check theGirl x and y Incremented distance
         xIncrement, yIncrement = theGirl.Distance_Moved()
         # Move all the tree sprites accordingly
         for tree in allTrees:
            tree.move( xIncrement, yIncrement )
         # If there Is a timePack move It as well
         if timePack is not None:
            timePack.move( xIncrement, yIncrement )
            if timePack.rectangle.bottom < 0:
               timePack = None
         distanceTraveled += yIncrement
```

Next handle the meat of the collision detection. Check all grouped tree sprites in the `timePack` using the `Collision_Watch` method:

```
# check for collisions with the trees
      treeBoxes = []
      for tree in allTrees:
         treeBoxes.append( tree.Collision_Watch() )
      # Retrieve a list of the obstacles colliding with the theGirl
      Collision = theGirl.Collision_Watch().collidelist( treeBoxes )
      # When colliding play a sound and subtract from the time left
      if Collision != -1:
         collisionSound.play()
         allTrees[ Collision ].move( 0, -540 )
         theGirl.Collision()
         timeLeft -= 5

      # Determine whether theGirl has collided with a timePack
      # A timePack must exist first
      if timePack is not None:
         if theGirl.Collision_Watch().colliderect( timePack.rectangle
):
            # Play a sound and Increase the time left
            chimeSound.play()
            timePack = None
            timeLeft += 5
```

There are only a few things left to do before yoou can exit the `while()` loop. First you want to draw any dirty or changed objects, mainly the trees and the `timePacks`. You also want to check to see if `theGirl` has reached the finish line, and, if so, exit the loop. Finally, you want to check the time; once `timeLeft` has reached 0 the game will also exit the loop:

```
# place objects on screen
      dirtyRectangles.append( theGirl.place( screen ) )
      for tree in allTrees:
         dirtyRectangles.append( tree.place( screen ) )
      if timePack is not None:
         dirtyRectangles.append( timePack.place( screen ) )
      # update whatever has changed
      pygame.display.update( dirtyRectangles )
      dirtyRectangles = []

      # check to see If we have reached the end of the course
      if distanceTraveled > COURSE_DEPTH:
         # Set a flag that says we have won!
         courseOver = 1

      # check to see If our time has run out
      elif timeLeft <= 0:
      break
```

Whew! Now, just a bit of wrap-up code at the end of `main()` and after exiting the `while()` loop. If you have exited the `while` loop and `courseOver` is set to 1, that means the player reached the end of the course and should get praise. Otherwise she lost.

```
if courseOver:
      applauseSound.play()
      message = "You Win!"
   else:
      gameOverSound.play()
      message = "Game Over!"
```

Of course, you use your handy-dandy `Display_Message` function to tell the player what happened:

```
   pygame.display.update( Display_Message( message, screen,
      background ) )
```

Use the event queue to wait for the player to gracefully exit the program:

```
   # wait until player wants to close program
   while 1:
      event = pygame.event.poll()

      if event.type == QUIT or \
         ( event.type == KEYDOWN and event.key == K_ESCAPE ):
         break
```

Finally, close off the `main()` function and make sure `main` is called with this typical end to the Python program:

```
if __name__ == "__main__":
   main()
```

Summary

Wow, you've come a long way. Just a few short 80 pages or so ago you were a newbie Python programmer; now you can surf with the best of them! You should feel comfortable creating a game loop, loading sounds and graphics, and doing basic networking with Python now.

Important points from this chapter:

- The two keys to Pygame are the `surface` and the `rect`.
- Really understanding blitting and sprites can greatly increase your game's performance.
- There is a ton of libraries that exist for doing things in Python.
- Tkinter has more methods and constructs than you can throw a stick at.
- Tkinter's `pack()`, `grid()`, and `place()` methods are the key to organizing the Tkinter GUI.
- There are tried and tested libraries for dealing with common development needs like networking and sound, most of which are uncomplicated.

Questions and Answers

1: Q: Why didn't you cover [popular game programming library]?

A: A: There is so much out in Python land that it would simply be impossible to include detailed references to everything that is out there. Not only is the amount of development work immense, it is constantly changing.

2: Q: Which graphics library is the best one to use for my first best-selling game?

A: A: Each library seems to have its own strengths and weaknesses. However, this important decision should be based on your project's needs, not on the features of any particular library. With the rapid change in today's technical world, I would also check and make sure a library has had recent updates and a number of faithful, experienced users before launching a project with it.

Exercises

1: Use `Load_Image` to create a simple slideshow that switches between images every few seconds or with a keyboard click.

2: What are the steps taken to create a simple game engine with Pygame?

3: Change the event code in `Monkey_Toss.py` or Snowboard! to take mouse input instead of keyboard input.

4: List at least three of the OSI network layers.

5: Alter `Load_Sound.py` so that it is capable of playing a MIDI, MP3, or any file besides a WAV.

Chapter 5. The Python Game Community

Even snakes are afraid of snakes.

——Steven Wright

Python's game-development community is extremely active, and literally dozens of prebuilt game engines are available through the GNU open-source community license. There are also specific tools and libraries for utilizing and creating art and graphics, not to mention resources for networking and massive multiplayer gaming. It is not possible within the confines of this book to list all of the active projects and awesome tools available to the young Python programmer; you just have to dive in and start researching. This chapter starts the process with tools and resources that I have had some good experiences with; I think it'll be a good place to start.

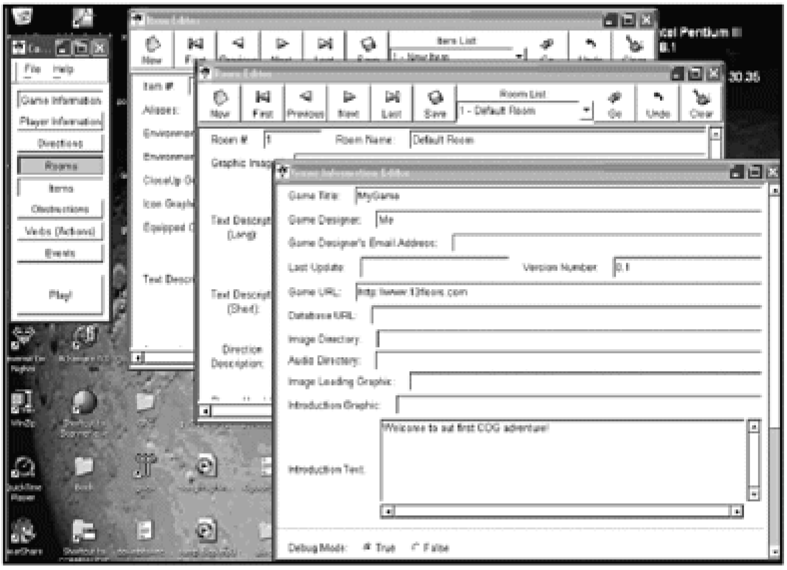
Engines

An engine is simply a tool, and this chapter focuses on tools and engines available that help you program games. These tools are all Python-based and open-source, and, for the most part, are geared towards the beginning programmer and so have easy-to-use interfaces.

The Cyclon Online Gaming Engine

The Cyclon Online Gaming Engine (COG for short) is an open-source computer game-authoring system. The system comes with a development application to facilitate game creation, a "fill-in-the-blanks" GUI that brings up windows in which you set up the game information, player information, rooms, directions, items, events, and even define action verbs that can be taken in by the text parser. The development application is shown in Figure 5.1.

Figure 5.1. The COG development application



COG currently supports a semi-Myst interface, with photo-realistic screens, text-based input, and mouse company-point movement. Games created with COG are meant to be run online via HTTP or through a Web browser interface. The engine can be found on Sourceforge at http://cogengine.sourceforge.net/.

Python Adventure Writing System

The Python Adventure Writing System (PAWS) is a text adventure system developed by Roger Plowman. As with many Python-based game tools, PAWS is aimed at the non-programmer and consists of a game engine, a world library, and a play module. PAWS is fairly well documented, and comes with a few sample games and two great explanatory texts, one aimed towards first-time game writers and another, aimed towards code-heads, that explains how the Python sources work. Even the source code

itself is well documented—especially the sample games, which read like tutorials themselves. You can find PAWS on the CD accompanying this book under the Python section (see Figure 5.2).

Figure 5.2. The PAWS engine at work



PAWS includes a few fun tricks to keep its games lively. For instance, it has a `say()` function that takes the place of `print` in Python; `say()` has the special ability to read commands for paragraph breaks, boldface, titles, and color. These tricks are especially helpful when designing a text-based game. `say()` also has a parse alias, Object, which is called `P` in the code for short. PAWS also includes a number of fun and unique classes, set up in the core game and in the universe library, for creating game objects and doing lots of useful things. These classes and what they are used for are outlined in Table 5.1.

Table 5.1. PAWS Classes

Class	Summary
`ClassActivatableItem`	Creates items a player can turn on and off
`ClassActor`	Creates people, animals, monsters, and other things players will be able to talk to or fight with. Based on `ClassBasicThing`.
`ClassBaseObject`	Base class for creating all "things" the player can interact with
`ClassBasicThing`	Defines basic physical laws. Base for specialized classes of "things"
`ClassContainer`	Creates containers that can hold things
`ClassDirection`	Used to create direction traveled by the player
`ClassDoor`	Creates one side of a door

Table 5.1. PAWS Classes

Class	Summary
`ClassFundamental`	The base root class of all the other classes. All other classes are based on this parent
`ClassGameObject`	Creates the game object
`ClassGlobal`	Creates a global object
`ClassItem`	Creates an object that can be taken and carried by the player
`ClassLockableDoor`	Creates a lockable door
`ClassMonster`	Defines anything with combat abilities. Based on `ClassActor`
`ClassOpenableItem`	Defines items that can open or close
`ClassParserError`	Stores error messages
`ClassPlayer`	Defines the player character object
`ClassRoom`	Defines room
`ClassScenery`	Creates props and atmosphere
`ClassShelf`	Creates a fixed shelf that items can be placed on
`ClassUnderHider`	Creates an item that drops contents when taken

PAWS makes it very easy to develop games quickly if you're accustomed to Python. For instance, the directions a player can traverse are set up through a Python dictionary. An example map might be something like the following:

```
MyRoom_1.Map = {North:      "You can't go that way.",
                   Northeast:  "You can't go that way.",
                   East:       "You can't go that way.",
                   Southeast:  "You can't go that way.",
                   South:      "You can't go that way.",
                   Southwest:  "You can't go that way.",
                   West:       "You can't go that way.",
                   Northwest:  "You can't go that way.",
                   Up:         MyUpstairsRoom_1,
                   Down:       MyDownstairsRoom_1,}
```

PAWS knows its maps well enough to figure out how to link rooms together or print out a string if that's what you want to have happen when a player travels in a certain direction. Items and rooms in PAWS are defined by using the class and then overriding the appropriate defaults methods, like so:

```
MyItem_1 = ClassItem("Mine")
MyItem_1.Bulk = 1
MyItem_1.StartingLocation = MyUpstairsRoom_1
```

In this example, I defined an item, `MyItem`. The argument given (`"Mine"`) is the noun descriptor PAWS will use to reference the item. The `Bulk()` and `StartingLocation()` methods (inherited from `ClassItem`) set up where the item will originally be found, along with its weight/size in the player's inventory.

Other fun PAWS features include a parser that can be extended so that a programmer can add new verbs and adverbs, game daemons that can be spawned to run functions at every player turn, and "fuses" that will run a function after a delay of so many turns. There is even a debug mode that allows you, for testing purposes, to trace commands and set variables while playing.

To get the latest version of Paws, hit Roger's site, at http://members.nuvox.net/~zt.wolf/PAWS.shtml.

PyPlace

PyPlace, by Peter Goode (and based on work by Pete Shinners), is a tool for generating isometric maps—"Place" rendering in Python.

The power behind PyPlace is a render.py module. This render model takes in a map object, which is basically a three-dimensional array, and uses the map to render an isometric view map with a number of square tiles (which are provided in a .png format).

Unfortunately, the project has been in alpha for quite a while, and it appears as though development on the project has stopped. Still, for the guru, this could be a good starting place for an isometric game engine. Find the project homepage at: http://www.mrexcessive.net/pyplace.

And find it at Sourceforge project page at http://sourceforge.net/projects/pyplace/.

Python Universe Builder

The Python Universe Builder is a set of Python modules used to create text-based games or works of interactive fiction. PUB was originally built by Joe Strout and was subsequently revised by Terry Handcock for his AutoManga project. PUB is currently now under the wing of Joshua Macy, who has made efforts to update PUB for Python 2.0 and document the project. The Sourceforge page can be found at http://py-universe.sourceforge.net/index.html.

NOTE

The Basic Universe Simulator

PUB's younger brother, the Basic Universe Simulator, is a set of Python code that demonstrates interactive fiction and Python. It is meant to be a short example of what Python is capable of, or a building block for a more complex game (the BUS is really just a few scripts capable of parsing English sentence-like commands). BUS was also built by Joe Strout and can be found at his Website, http://www.strout.net/.

PUB has a handful of modules for importing, as shown in Table 5.2. The modules are object-oriented, and have several big base classes grouped around objects that players interact with and verbs that the PUB uses to translate player commands.

Table 5.2. PUB Modules

Module	Function
`demo`	Contains a simple demo game
`gadgets`	For building specific objects
`pub`	Contains globals
`pubobjs`	Contains standard objects
`picklemod`	For saving (pickling) entire modules
`pubscore`	Contains datatypes, functions, and constants
`pubtcp`	Used for network support
`pubverbs`	Contains standard verbs
`tcpdemo`	Used for MUD adaptation

PUB also has classes for schedulers, commands, the parser itself, and events, which can be used to create everything used in the engine.

After importing PUB, you can begin building MUD-like rooms and areas fairly quickly.

```
# Create a room with module pubobjs and method Room
MyPrisonRoom = pubobjs.Room("Dungeon Prison Cell")
# Describe room with desc method
room.desc = "You're in a small barred cell with walls of stone.\
To the north is a rusty Iron-barred door. \
A small bowl filled with water lies In one corner of the room."
# Establish north exit exits
room_n= Exit("north,n,out,bars,door")
# Describe exit
room_n.desc = "The door appears to be unlocked."
# Add object Into the room
water = pubobjs.Liquid("water,liquid")
# Describe object
water.desc = "It appears to be ordinary water, and fairly clean."
```

This example first creates a sample room called `MyPrisonRoom` using `pubobjs.Room`, and then describes the room and establishes exits with the `desc()` and `Exit()` method calls. Then an object, in this case a `Liquid()` object, is created within the room and described in a similar way. Notice how creating an object in a room and creating the room itself are nearly identical.

PUB's biggest strength is likely its sentence parser, which allows fairly complex input from players ("Get the dragon and put it in the shoe…").

The Sourceforge page provides a few sample games (including a sample game that has been turned into a MUD version) and a template script that handily shows, via comments, where objects must reside. The source code itself is also available and fairly well commented.

The Pyzzle Game Engine

Pyzzle is a free (under the GNU public license), pre-built game user-interface in the spirit of the Myst and Riven (it is included in this book's CD under the section on Python). Authored by Andrew Jones and written in Python and Pygame, the engine includes the following features:

- A modular rendering interface capable of using OpenGL, SDL, or Direct3D.
- Runs on several platforms (Windows, NT4, OSX, BeOS, FreeBSD, IRIX, and Linux).
- Full API using Python scripting.
- Support for different display sizes (640x480, 800x600, 1024x768, and so on).
- Ambient sound, music, and sound effects (using WAV files).
- Over-slide images (formats include BMP, GIF, PNG, JPG, PCX, and TGA), text using True Type fonts, and movie playback using MPEG files.
- In-game objects that players can carry.
- Zip navigation option.
- Customizable color graphical cursors.
- Slide-like Riven-style area transitions.
- Basic menus.

See Figure 5.3 for a look at Pyzzle's packaged demo game in action. Pyzzle is composed of the handful of modules listed in Table 5.3.

Figure 5.3. The Pyzzle demo game showing the engine at work



Table 5.3. Pyzzle Modules

Module	Use
`AmbientSound`	Defining ambient sounds and music
`Image`	Defining over-slide images
`Movie`	Defining movies
`Object`	Defining objects
`parameters`	Controlling the global game parameters
`paths`	Defining the default paths
`Pyzzle`	The base engine
`Slide`	Defining slides. The basic graphics unit
`Sound`	Defining sound effects
`Text`	Defining over-slide text

The API isn't quite finished as far as documenting goes, but just opening up the demo game files (check demogame.py) and perusing them can be quite revealing (and, of course, the source code is freely available).

Once Pyzzle has been imported, you use `parameter` methods to set the game parameters like screen size and background color:

```
#import Pyzzle
import pyzzle
from pyzzle import *

# Set a few game parameters
parameters.setScreenSize((800,600)) # window size in game
parameters.setBackgroundColor((0,0,0)) # set background to black
```

Then you use the `paths` module to set the paths to the WAV, MPEG, screen, and other files:

```
#Tell Pyzzle about a few paths to use.
paths.setSlidePath(os.path.join('data', 'slides'))
paths.setSoundPath(os.path.join('data', 'sounds'))
paths.setImagePath(os.path.join('data', 'images'))
paths.setMoviePath(os.path.join('data', 'movies'))
```

You will need at least one slide, which is basically a game screen. It can be easier to start off giving each slide a label:

```
#define Slide containers
MyStartingSlide   = Slide()
MySecondSlide  = Slide()
MyThirdSlide   = Slide()
```

And then defining each slide:

```
# Define slides
MyStartingSlide.setNavType(standard)
MyStartingSlide.setSlideFile('MyImageFile.jpg')
MyStartingSlide.setNavigation([MySecondSlide, MyThirdSlide,])
MySecondSlide.setNavType(standard)
MySecondSlide.setSlideFile('MyImageFile2.jpg')
MySecondSlide.setNavigation([MyStartingSlide])
```

This would set and connect two different images as slides that could navigate to each other. Starting the game up requires two lines:

```
# Set the starting slide
pyzzle.setFirstSlide(MyStartingSlide)
#start the game
pyzzle.start()
```

There is a lot more that Pyzzle can do. Each slide can include music, items, special effects, and special behavior defined for clicking and navigating. Text, objects, ambient sound, containers, and puzzle control logic can all be defined and used to make a great game. For the latest version of Pyzzle, check out its homepage on Sourceforge, at http://pyzzle.sourceforge.net/.

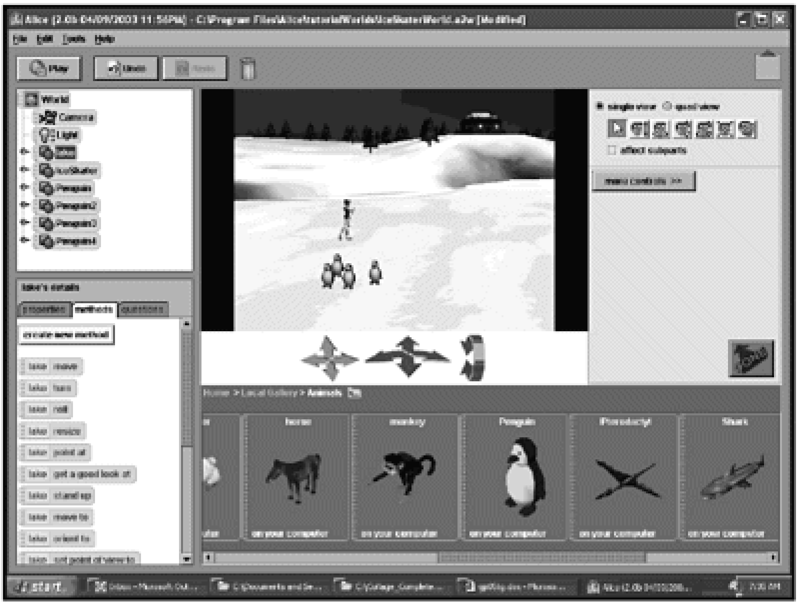
Graphics

What do 3DS Max, Lscript, Lightwave, Alice, Maya, Blender, Animation Master, TrueSpace, RenderMan, and Poser all have in common? Well, besides being graphic programs and 3D applications, they are all Python scripting interfaces. Python is ideal for the struggling artist; it's able to link up to industry gear and is perfect for creating quick custom tools or automating repetitive tasks.

Alice

Alice is a tool for developing three-dimensional graphics, built around the concept of "3D for everyone." Most 3D engines require the programmer to know extensive trigonometry, vector algebra, and other painful math. Alice is designed to provide non-programmers with access to 3D programming and interactive worlds. One of the things that makes Alice powerful is that it has a very straightforward, easy-to-learn GUI (shown in Figure 5.4) for placing, sizing, tweaking, and animating three-dimensional objects and spaces.

Figure 5.4. The Alice GUI



Alice is open source and made available by its current developers and copyright holders, the Stage Three Research Group at Carnegie Mellon University, and can be found online at http://www.alice.org.

The worlds and content created with Alice are freely distributable, as long as the stipulations in the license are followed. The Alice project initially began at the University of Virginia, and over the years has received support in the form of grants from DARPA, Intel, Microsoft, NSF, Pixar, Chevron, NASA, the Office of Naval Research, Advanced Network and Service Inc., ONR, and the Python community itself.

Currently Alice supports two-dimensional graphic imports (via drag and drop or through its built-in billboard) and .ase files, which are ASCII Scene Export files used for exporting 3D wire-frames on several 3D modelers (including 3D Studio Max). Alice is also capable of importing music and sounds by using MP3 files. The engine comes equipped with hundreds of models and sounds pre-built and packaged for the newbie.

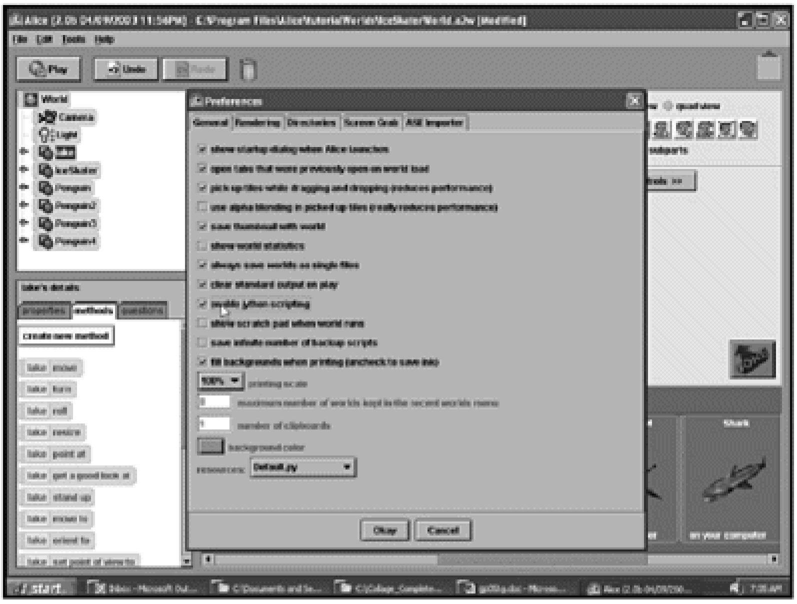
Alice actually has draggable programming constructs (for example, `if/else` statements and loops) that can be used to set the behavior of the models. Underneath the GUI is a complete language that supports methods, arrays, lists, functions, recursion, and so on.

Alice has recently gone through a complete re-development, and work is ongoing to allow Alice to export and import more formats and run on more platforms. Originally, Alice was completely Python—the core, the code, the whole enchilada. With the recent major rewrite (which has been ongoing since 1999), much of the software has been rewritten in Java. However, the engine is still scriptable via Jython.

Jython is an implementation of Python. However, Jython is written completely in Java, and is integrated into Sun Microsoft's Java 2 J2EE platform. This means Jython has all the dynamic object-oriented features of the Python language, and also runs on any Java platform.

In order to implement Python/Jython scripting in Alice, you need to first enable it. You can turn on Jython scripting under the Preferences menu. Select Edit, Preferences, Enable Jython Scripting, as shown in Figure 5.5.

Figure 5.5. Enabling Jython scripting in Alice's GUI



Once scripting is enabled, every object within the Object Tree (the top left-hand window, which includes any instance of three-dimensional objects, including the world itself) is script editable with a right-click of the mouse, or through one-line scripts via a "go" executable line (see Figure 5.6). You can also access scripts when editing methods

(Alice has a built-in method editor) with two draggable tiles called Script and Script-Defined Response.

Figure 5.6. Editing a penguin object script from the object tree



The Script tile allows you to type in code that will be run when that script method is run in the Alice engine. The Script-Defined Response is used to fire pre-composed Alice animations.

Objects in Alice can be called, using their names, through scripts, and their properties and variables are accessed just like member variables:

```
Penguin.isShowing = false
```

All of this is pretty powerful—not only can you script objects via Python/Jython, but with Jython you also have access to the entire Java API. The scripts can also call built-in Alice animations and Alice's "RightNow" methods, like those outlined in Table 5.4.

Table 5.4. Alice's RightNow Methods

Method	What it does
`DoInOrder()`	Runs a series of animations
`IfElseInOrder()`	Runs animation list if the condition is met for `if/else` statements
`isShowing()`	Sets subject to be visible or not visible
`ForEachInOrder()`	Iterates through a list
`MoveAnimation()`	Moves subject

Table 5.4. Alice's RightNow Methods

Method	What it does
`moveRightNow()`	Moves subject immediately if given direction and amount
`PositionAnimation()`	Sets subject position in world
`ResizeAnimation()`	Resizes subject
`resizeRightNow()`	Resizes subject immediately
`rotateRightNow`	Rotates on given axis immediately
`setOrientationRightNow`	Sets subject's orientation via 3D matrix immediately
`SoundAction()`	Plays given sound at specified volume
`TurnAnimation()`	Rotates subject
`turnRightNow()`	Rotates subject immediately given amount
`WaitAction()`	Waits for given duration
`WhileLoopInOrder()`	Runs through animation list while condition is true

These methods (and many others—check out the Alice2 documentation) can be called on models within Alice, but also on Alice's camera (the "watcher" point of view) and other objects like lights.

Let's say you wanted to define an animation function in Jython. You can define the animation just like you define any other function:

```
def MyAnimation(MyObject):
        return MyAnimation
```

In this case, the function `MyAnimation` will take in `MyObject` as an argument and send back `MyAnimation` as the animation series you want the model to execute (assuming that the object will be an Alice model). Now let's set the animation to do something:

```
def MyAnimation(MyObject):
        turn = TurnAnimation(MyObject, right, amount=1.0)
        move1 = MoveAnimation(Forward, amount =1.0, duration =1.0)
move2 = MoveAnimation (Backward, amount=1.0, duration=1.0)
MyAnimation = DoInOrder(
        MyObject.IsShowing = true,
        move1,
        turn,
        move2,
)
        return MyAnimation
```

You define `move1` and `move2` to move forward and backwards using Alice's `MoveAnimation` method. Then you set `turn` to give the model a spin using

`TurnAnimation`. Finally, you make sure the object is visible with `MyObjectIsShowing` and run your series of animations.

AutoManga

Although now nearly defunct, AutoManga is a solution for digital cell animation. Japanese Manga-style animation is the idea behind AutoManga, and the engine is implemented with Python scripts that call C/C++ extensions for SDL routines. The engine was developed by Terry Hancock, has had a number of other contributors over the years, and originally was to be connected to the Python Universe Builder to handle interactive fiction and use XML for sequencing resource files.

Much of AutoManga was completed, including lighting effects and the ability to pull a few different formats for background images and animation cells, but the project unfortunately hasn't seen much action in the past year or two. Still, it is a good starting point for frame and cell based Python animation; the developer notes and files are located on Sourceforge, at http://automanga.sourceforge.net/.

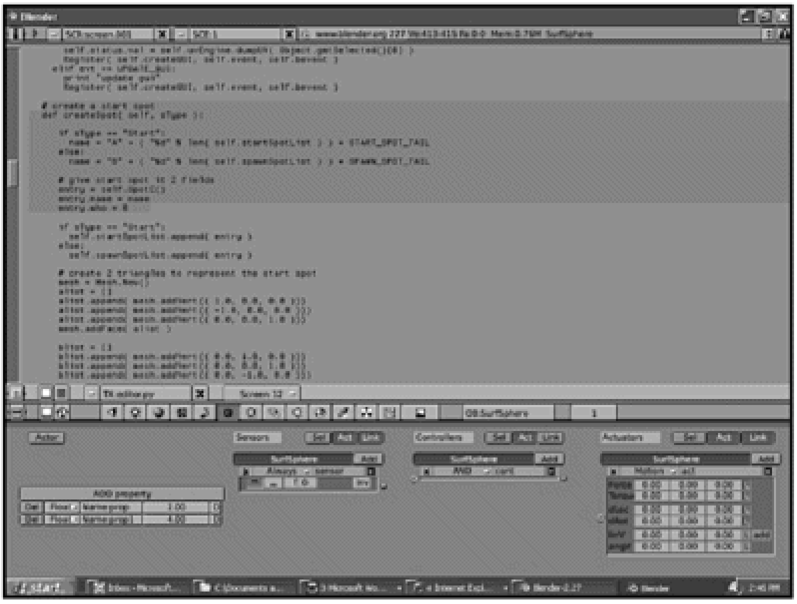
Blender

Blender is a 3D graphics suite with a tumultuous history. Originally, Blender was a rewrite of the Netherlands animation house NeoGeo's 3D toolset. One of the co-founders of NeoGeo, Ton Roosendaal, also founded a spin-off company called Not another Number (NaN). This company's model was to further develop and market Blender technology. Initially this company faired very well, raising millions of dollars and gaining thousands of customers, but it was hit with hard economic times. In 2001, the company announced bankruptcy and the investors closed down NaN.

Blender, however, proved to have a strong will to live. Roosendaal started a non-profit foundation and began the "Free Blender" campaign with the idea of opening up Blender to the community as an open-source project. He worked with NaN's investors to agree to a plan wherein the Blender Foundation would be able to purchase the intellectual rights and source code of the Blender engine. Then, to the surprise of everyone, Roosendaal and several ex- NaN employees, with the help and support of Blender's loyal users, managed to raise 100,000 EUR in seven weeks to make the purchase. Blender was free, and continues to be free to this day, supported by developers and used by artists around the world, under the GNU GPL License.

Blender can be used for 3D modeling, animation, game-engine scripting (in some versions), and rendering. Most useful is Blender's built-in text editor (see Figure 5.7) for Python scripts, which can be used to customize tools, set up animations and effects, and even build sophisticated AI control over lighting and game objects.

Figure 5.7. Blender's text editor readily opens a Blender Python script



Blender offers a number of Python modules (shown in Table 5.5) to use in scripting. Some of them are still being ported into the newest version of Blender as of this writing.

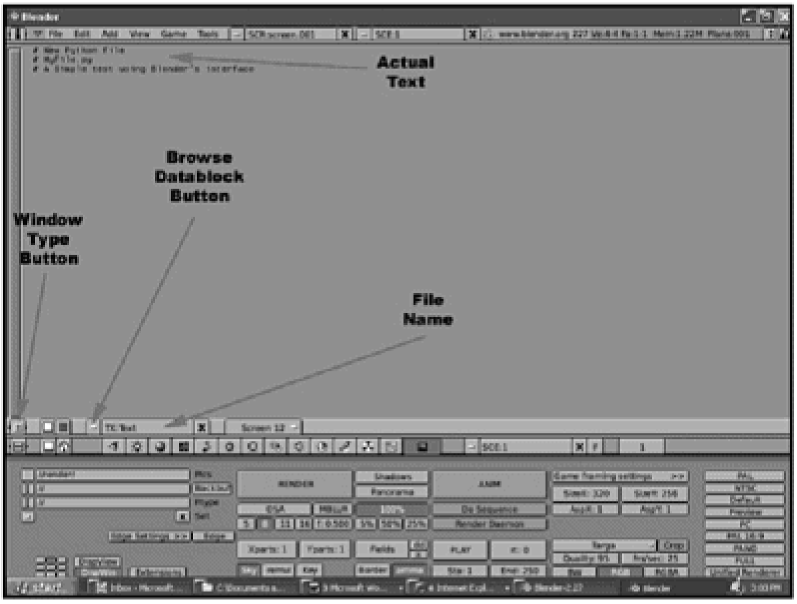
Table 5.5. Blender Python Modules

Module	Description	Porting Complete
`Blender`	The main Blender module	yes
`BGL`	The Blender OpenGL module	yes
`Camera`	The Camera module	yes
`Draw`	Display module	yes
`Image`	The Image module	yes
`IPO`	The IPO animation key module	no
`Lamp`	The Lamp module	yes
`Material`	The Material module	no
`Mesh`	The Mesh module	no
`Nmesh`	Low level mesh access	no
`Object`	The Object module	no
`Scene`	The Scene module	no
`Text`	The Text module	yes
`Window`	The Window module	yes

To switch to the scripting mode in Blender, press the Shift and F11 keys simultaneously or go to the current Window Type button and choose Text Editor. Click the Browse

Datablock button and choose Add New Blender to open a blank .py file. Blender will automatically name the file TX:text; you can change the name by clicking on it and typing in the new name (see Figure 5.8).

Figure 5.8. Highlighted text controls in Blender



To test out Blender, start by renaming a text file to MyFile.py, and then import the main Blender module. From that point on you have access to the Blender methods such as `Object`:

```
import Blender
MyObject=Blender.Object.Get("Some_Object")
```

When running scripts on objects in Blender, you would normally have two windows open. One would be a workspace with the object within, and the second would contain the Python script that you would run on the object.

Let's say you needed to run some complex math on a `Mesh` or `Nmesh` in Blender. First you import `Mesh` or `Nmesh`:

```
import Blender
from Blender Import Nmesh
```

Then grab the `mesh` object, its name, and its raw data using `Object` and `Nmesh` methods:

```
MyObject=Blender.Object.Get("Some_Mesh_Object")
MyMeshName=MyObject[0].data.name
MyMesh=Nmesh.GetRaw(MyMeshName)
```

Finally, run your complex math on each vertex, replace the values in your objects, and have Blender redraw the object:

```
for each_vertex In MyMesh.verts:
        # complex math here
        # complex math here
        # complex math here
Nmesh.PutRaw(MyMesh, MyMeshName)
Blender.Redraw
```

Blender a is an excellent demonstration of the power of open source and open community development. Blender's user base is extremely supportive and creative, and is busily at work at making Blender the best appliance since toasters. You'll find information on Blender at

- **The Blender community site.** http://www.blender.or.g
- **The Blender foundation site.** http://www.blender.org/bf.
- **The Blender release page.** http://www.blender3d.org.

Nebula

Nebula is an open-source, 3D, real-time, multi-platform game engine that supports Direct X and OpenGL. The project is brought to us by the game studio Radon Labs, in Berlin. Nebula is actually implemented with C++, but what makes it super-fun is that it is also scriptable with Python, Lua, and Tcl/Tk. I'll talk a bit more about Nebula later on in this book (specifically in the Lua sections).

Panda3D

Panda3D is a rendering engine for SGL. The core of the engine is in C++, but Panda3D also provides a Python scripting interface and utility code. I'll talk a bit more about Panda3D in the section on commercial games later in this chapter.

Poser

The Poser Pro Pack and Poser 5 come equipped with Python scripting as an available resource for artists; this is mainly used to automate advanced functions in the interface. Python scripts can be accessed from Poser's Window menu, which opens up a Python Scripts dialog box, as shown in Figure 5.9.

Figure 5.9. Accessing Poser's Python Scripts dialog box



The dialog box can be used as a placeholder for commonly used scripts. Clicking on a script with the Alt key on a PC or the Control key on a Mac will bring up a text version of the script that you can edit.

When creating custom scripts, much of the work on Poser is done through the `Scene`, which is part of the Poser import module:

```
# First Import Poser module
import poser
# Create a scene
MyScene = poser.Scene()
        # Then you would do things to the poser scene
# And at the end re-draw the scene
Myscene.DrawAll()
```

Pretty nifty, huh? Poser actually has a very deep API for interacting with Python; it goes way beyond scenes and comes equipped with pre-defined scripts for you to use. There is also a fairly large knowledge base and plenty of sample scripts within the community.

Information on Poser can be found at Curious Labs's site, at http://www.curiouslabs.com/products/poser4#productinfo.

Commercial Games

Game engines and graphics are all well and good, but what about actual commercial games, you ask? You're in luck, for Python has slithered its way into many a shop. The language has been used as the primary scripting tongue for quite a few major games, and there are also a handful of game development tools, scriptable via Python, that have also been released.

Eve Online

Eve Online is a massive multiplayer online game that won the award for best online game in Game Revolution's The Best of E3 2002 Online Awards, and was also featured shortly after its release at 2003's E3 conference. Created by Iceland's CCP Games and released in 2001, Eve's world is a massive RPG science-fiction environment featuring photo-realistic graphics and a real space-faring feel.

What makes Eve special for us is that its game-logic is controlled by Stackless Python. CCP used Stackless on both the client and server side to free its programmers from many of the mundane tasks of model behavior and instead focus on the creative parts of AI. Stackless also allows CCG to easily make changes to the game and game behavior, even while the game is running, which is extremely important for its persistent online world model.

Freedom Force

Freedom Force, a popular super-hero multiplayer game from Irrational Games, was nominated for handfuls of PC Gamer's annual 2002 awards, and Irrational is currently working on an expansion of the game. Irrational used NDL's NetImmerse game engine and Freedom Force was co-published by Crave Entertainment and Electronic Arts. Many of the game's functions were exported to the Python side, so that Python could set and move objects and control camera movements. The single-player levels were scripted with Python as well, in order to control mission control and cut-scenes.

Python was used with custom extensions provided by the Freedom Force engine, and the key to using these extensions is understanding the scripting guides, which you can download from Irrational games at http://www.irrationalgames.com/modforce/Editor/script.htm.

Freedom Force launches two Python scripts (located in its system folder): startup.py and init.py. Both of these files are used to set the data paths for the game; by adding to the default path, you can change which module `ff` (Freedom Force) loads up at the beginning:

```
import ff
ff.DefaultPath = "MyModule;data"
```

Python scripts control the flow of a module or adventure and can be used to script missions, create events that spawn new enemies, check for mission success and failure, trigger speech, and run cut-scenes. Each mission has a single script file (called mission,

py) with which it is associated and must be in the same folder as mission.dat (this file is commonly know as a mission script).

There are also level offshoots, called briefings and intermissions, that are loaded in between missions. These are scripted in the same way as missions but use a base.py file and a base.dat file instead.

The custom extensions provided by the Freedom Force engine are huge. Everything from AI to Object control to missions to camera movement is completely accessible via the Python scripting interface. Let's take a look at one example, a cut-scene snip from Freedom Force. The Freedom Force camera has a number of methods for using cut-scenes, as illustrated in Table 5.6.

Table 5.6. Freedom Force Cut-Scene Methods

Method	Purpose
`play()`	Plays a cut-scene
`isPlaying()`	Determines whether a cut-scene or scripted sequence is currently playing
`startCS()`	Starts a cut-scene
`endCS()`	Ends a cut-scene
`endBriefingCS()`	Ends a briefing
`startCSNormalScreen()`	Starts a cut-scene but doesn't go into widescreen mode
`isCSPlaying()`	Returns true if a cut-scene is currently playing
`playTransition()`	Plays the logo transition

Using these methods to start and stop a cut-scene would look like the following:

```
# Define Cutscene
MyCutscene = [
(
# Start Cutscene
  "startCS()",
)
# End Cutscene
  "endCS()",
)
```

Those who have been paying attention will notice that cut-scenes in Freedom Force are Python lists; here is the same code condensed to one line for familiarity:

```
MyCutscene=[(item1,)(item2,)(etc)]
```

Later in the code you call the `play()` function and viola! The `MyCutscene` cut-scene would run:

```
play(MyCutscene)
```

Of course, this cut-scene doesn't do much at all, but that's where FF's camera controls come in. The camera is enabled by a `Camera_LookAtObject()` command and released back to the player with the `Camera_Release()` command. `Camera_LookAtObject()` can be set with a number of commands common to the FF camera, as shown in Table 5.7:

Table 5.7. Freedom Force Camera Controls

Command	Description
`objectName`	The object to track
`camDist`	The zoom distance
`camPitchRot`	Angle of pitch around object right vector, in degrees
`camYawRot`	Angle of yaw around object up vector, in degrees
`camSpeed`	Time in seconds it will take to complete the move
`movePathMode`	Set camera snap (`CPM_SNAP`, `CPM_SCROLLTO`, `CPM_HOMING`, or `CPM_SIMPLEPATH`)
`camAction`	Set camera move (`CA_MOVE`) or tracking (`CA_TRACK`)
`callbackFunc`	Sets a Python script function to call when finished
`fUser`	User defined data

Given the camera controls in Table 5.7, you can move the camera around the main player or protagonist:

```
MyCutscene = [
(
  "startCS()",
  "Camera_LookAtObject('My_Player',-
195,30,384,3,CPM_SCROLLTO,CA_MOVE)",
  "Camera_LookAtObject('My_Player',-
200,20,320,3,CPM_SCROLLTO,CA_MOVE)",
)
  "endCS()",
)
]
```

Table 5.7. Freedom Force Camera Controls

Command	Description
`objectName`	The object to track

Table 5.7. Freedom Force Camera Controls

Command	Description
`camDist`	The zoom distance
`camPitchRot`	Angle of pitch around object right vector, in degrees
`camYawRot`	Angle of yaw around object up vector, in degrees
`camSpeed`	Time in seconds it will take to complete the move
`movePathMode`	Set camera snap (`CPM_SNAP`, `CPM_SCROLLTO`, `CPM_HOMING`, or `CPM_SIMPLEPATH`)
`camAction`	Set camera move (`CA_MOVE`) or tracking (`CA_TRACK`)
`callbackFunc`	Sets a Python script function to call when finished
`fUser`	User defined data

Not bad for a quick delve into the Freedom Force API—and we've really just begun. There are actually a number of other camera commands to set wide-screen, introduce camera jitter, snap to objects or markers, fade in and out, and so on. Outside of the camera there are whole suites of functions and methods to set up narration, music, and sound effects, control NPCs and characters, set mission objectives and game flow, and so on and so on.

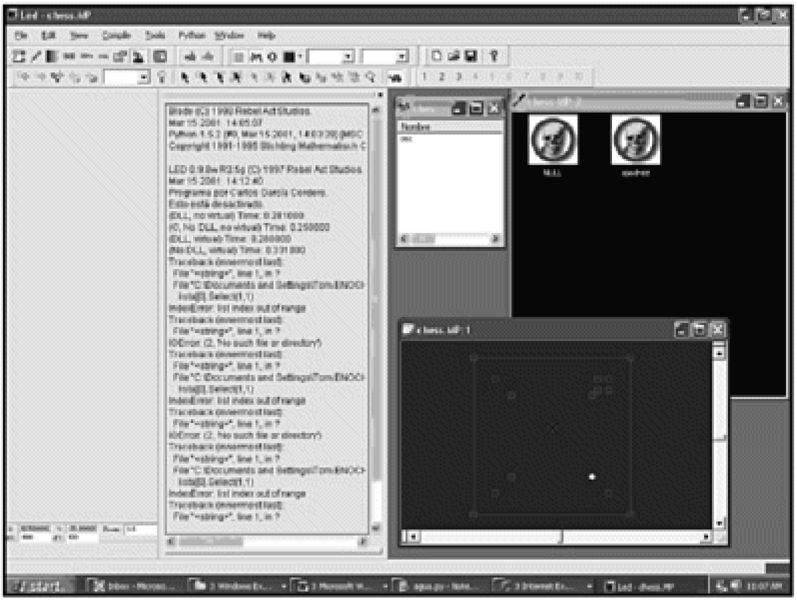
Severance: Blade of Darkness

Severance: Blade of Darkness is a fantasy combat game from Codemasters / Rebel Act Studios (which is now defunct). It is a mature-audience game released in 2001 along with a level editor (called LED) and a set of tools (called RAS) for making levels and mods, which are, of course, based on Python and wholly scriptable. A Blade of Darkness level generally includes:

- A .bw file which has the map architecture details, compiled from the LED map editor (uncompiled maps are .mp files).
- .mmp files, which are files with the textures used in and on the map.
- One or more Blade of Darkness (BOD) files that define the objects and characters that inhabit the mod.
- A number of Python scripts that initialize and make objects and npcs and so on.
- A level file (.lvl) that loads things up to the game engine (the .mmp bitmaps and the .bw map file).

The LED editor is shown in Figure 5.10 (notice Python on the top toolbar).

Figure 5.10. The LED editor with an open sample file



In the Python scripts, you'll find that objects (weapons, torches, and so on) are usually defined with a objs.py file, players with a pl.py file, configurations with a cfg.py file, the placement of the sun and its position with a sol.py file, and any water coordinates with a agua.py file.

Take a look at a sample agua.py file:

```
import Bladex

pool1=Bladex.CreateEntity("pool1","Entity Water",72000,39800,-2000)
pool1.Reflection=0.9
pool1.Color=90,20,20

pool2=Bladex.CreateEntity("pool2","Entity Water",116000,39800,54000)
pool2.Reflection=0.1
pool2.Color=60,10,10

pool3=Bladex.CreateEntity("pool3","Entity Water",116000,39700,46000)
pool3.Reflection=-0.5
pool3.Color=0,0,0
```

First, the necessary Bladex libraries (which hold most of the necessary commands and functions) are imported. `CreateEntity` is then called on to create three separate pools of water at three separate locations. Once instantiated, each pool is then further defined with the `Reflection` and `Color` methods.

NOTE

A handful of developers from Rebel Act started their own company called Digital Legends Entertainment at http://www.digital-legends.net/ shortly after RAS closed its doors. They are currently focused on producing their first game, Nightfall Dragons at http://www.nightfalldragons.com.

ToonTown

ToonTown, an online cartoon style mulit-player game, is the latest from the Walt Disney Imagineering studio. Players create their own cartoon avatars and explore a rich world where they can meet and interact with other "toons," earn jelly beans to put in the bank, and buy things (like a toon house or items for a toon house). There is even a bit of conflict thrown in, in the form of a "Cog Invasion" that is threatening the city.

Disney's ToonTown uses Python in a direct and powerful way. The ToonTown executable actually calls Python on the client when the program is instantiated. Python was also used in development of the game, particularly in the Panda3D rendering engine.

Panda3D is powered by Python, DirectX, and the Fmod music and sound effects system. After being used to create Disney's ToonTown it was released to the open source community and is currently under even more extensive development by both the VR Studio and the Entertainment Technology Center at Carnegie Mellon University. ETC is working on making a simple installer for Panda3D (the current installation is somewhat of a bear…ahem), creating solid documentation, adding to the basic model import functionality, and creating tools like level and script editors.

Note that there are two versions of Panda. One is the original release to the community from Disney, located on Sourceforge and found there at http://sourceforge.net/projects/panda3d/).

The second version is the release from Carnegie Mellon's ETC, and can be found online at http://www.etc.cmu.edu/projects/panda3d/downloads.

Panda is capable of importing Maya and 3D Studio Max models, as well as the standard .gif, .tiff, and .jpeg image formats. It has a fairly extensive API that is still undergoing documentation. It can also be extended with the SDK, and the engine itself is tweakable, as the code has been released to the community.

The two most important lines in any Pythoned Panda script are

```
from ShowBaseGlobal import *
```

and

```
run()
```

The first line imports the necessary Panda files (which takes quite a bit of time) and the second line runs the environment. Running these two lines in a script after installing Panda will create a large, blank, gray window. These two lines are the minimum needed to create a Panda environment.

Panda3D is built around the scene-graph, which is a tree-like object hierarchy structure. Individual objects, which are normally 3D models or GUI elements, are called

`NodePath` objects. `NodePath` objects inherit behavior from their parents, and there are a number of built-in, base, pre-defined `NodePath` objects in Panda.

Panda3D models are either .egg or .bam. EGG files are ASCII format (and therefore readable by humans), and .bam is their corresponding binary format (for faster load times). You load a 3D object in Panda using its global `loader` object, like so:

```
My3Dobject = loader.loadModel("3Dobject.egg")
```

All loaded objects in Panda are, by default, hidden from view. To change this, take the loaded object (which is now a `NodeObject`) and change its parent to `render`; doing so will make the object render onscreen:

```
My3Dobject.reparentTo(render)
```

Once the object is loaded, you can call upon all sorts of fun methods to manipulate it, from setting the x,y, and z coordinates with `setX()`, `setY()`, `setZ()` or `setPos(x,y,z)`:

```
My3Dobject.setX(4) # Moves the object 4 "feet: on the X coordinate
```

to changing the heading, pitch, and roll with `setHPR(heading, pitch, roll)`:

```
My3Dobject.setHPR(50, 30, 0) # Changes the model heading by 50 degrees
and pitches the
model upward 30 degrees
```

to changing the object's scale with `setScale()`:

```
My3Dobject.setScale(10) # sets the scale uniformly x10 in each
direction (x,y, and z)
```

Panda is also capable of handling events (mouse clicks and key presses), has a GUI system for creating UI elements like buttons and dialog boxes (which can be bound to Python functions), and can incorporate sound effects and music.

Beyond Python

So what else is there besides games and graphics? Well, a whole heckuva lot, actually. Being the adaptable language that it is, you'll find Python sunning on rocks and slithering in the grass just about everywhere on the planet. The projects in this section may particularly pique your interest.

Beyond

One problem with 3D titles is the massive amounts of knowledge and work required to design, maintain, and update them. Another is the constant re-engineering each independent gaming company must fund and support in order to create the latest and greatest. Beyond is a reusable object framework for game design that was created to address these problems. The idea behind the Beyond project was to identify which parts of the process are works that could be reusable, and then wrap them up as components in order to create robust, easily modifiable 3D games. Python was chosen for this project because of its adaptability to multiple platforms, and its extensibility.

The first version of Beyond, Beyond 1, was the development project and platform for UO2, which was to be an imaginary-planet, massively multiplayer game released by Origin Systems. Based on the Ultima fiction originally created by Richard Garriott, UO2 had player avatars with highly customizable identities capable of interacting with objects and other players in a massive world. Unfortunately, UO2 was dropped and never actually saw the light of day, but this has been a minor setback for one of the principle developers, Jason Asbahr, who now leads an open-source, virtual-world, MMP Python framework, Beyond 2.

Beyond 2 is still very young, with only Version 0.0.1 released, but, of course, it is being built on the backs of several other highly successful platforms, including the Nebula Device by Radon Labs, Beyond 1, and Twisted Python.

Technically, the original Beyond 1 project was based on many languages, but Python had a particularly interesting role. Client-server information is encrypted and passed through constructs called `SimObjects` using remote method invocations in Python.

The `SimObject` was a root object, or superclass, for all other objects. SimObjects were organized in an object-oriented hierarchy, and could perform actions by executing methods on themselves or other `SimObjects`.

Beyond 1 was also data-driven and had a master game database. World builders were able to alter and expand the world by adding behaviors, entities, and data into the database without changing the actual runtime code. Clients (players) would connect to local Python area servers which eventually connected to a main data base server and the main game database.

Sound libraries, network communications, and graphics were all wrapped into Python as extensions using SWIG (short for Simplified Wrapper and Interface Generator; more on SWIG in Chapter 12). Functions in C and C++ are exposed as Python function objects.

Progress on Beyond 2 and Jason Asbahr's other projects can be found at his Website, at http://www.asbahr.com/index.html.

The site includes several papers he has presented on Python and Python games. This set of papers also includes a port of Python onto the PlayStation 2 and Nintendo platforms.

Pippy

Pippy is a port of the Python language to the PalmOS currently under development at Endeavors Technology. Although still young, the latest version runs on Palm OS 3.5 or higher. A handful of the standard Python modules have been removed to reduce the code footprint, though these removals are mostly code features (like dynamic linking libraries) that aren't necessary on a Palm platform. Pippy can be freely distributed as long as the copyright notice is included; the latest versions can be found on its Sourceforge page, at http://pippy.sourceforge.net.

A handful of Python features have been removed for smooth running on the Palm, including the following:

- Floating point numbers/objects.
- Complex numbers/objects.
- Python parser and compiler
- Documentation strings
- Dynamic linking
- Signals
- Path-related code
- File I/O (`stdio` and `stderr` are simulated)
- Most of the Python library modules
- Most of the Python extension modules

Pippy does include a version of the popular Python interactive interface and a keyword popup menu interface with both a Keywords and Modules menu that contain built-in Python names, reserved keywords, and a listing of the built-in and extension modules.

Development work may still be needed on Pippy to reduce the code footprint, and currently Pippy works on a reduced version of Python 1.5.2. There are a few issues to work out with the Palm's stack (work is underway to bring Stackless Python to Pippy) and Palm's dynamic heap, but the early project results appear promising, the key being an active community willing to take the project to the next level.

Stackless Python

Stackless Python is a development effort led by Christian Tismer, and is a Python variant that doesn't use the C stack. The Python interpreter is written in C, so at some level every Python action is executed via C. Mostly this is good, but sometimes having multiple instances of Python C code running on the stack can cause problems, for example with recursion and with object references that build up on the stack.

Stackless has received quite a bit of community support and has been highlighted at a number of Python conferences. Several companies, including Twin Sun, IronPort, and

CCP Games have used Stackless in development. Stackless is a super-tool for Python work using co-routines or micro-threads; the popular MMOG Eve Online is a good example of Stackless use in this case. Stackless has gone through a few variations, and Tismer continues to maintain, update, and further improve the concept, tirelessly making Stackless faster, more portable, and efficient.

You can find more information on Stackless at Christian Tismer's Website, at http://www.tismer.com/.

Twisted

Twisted began its existence as an open-source, massive, multi-player game called Twisted Reality. Since then Twisted has become a way to create network applications, from network transports and protocols to secure client servers. Twisted is no longer just a toy. It is a competitive production server system, designed with a small footprint to run on low-end hardware and still be capable of handling thousands of users.

Twisted supports the following:

- Win32 events
- GUI (GTK, Qt, wxPython, Tkinter, and so on)
- TCP, SSL, UDP, Multicast, and UNIX sockets and subprocesses
- Scheduling
- Threading integration
- RDBMS event loop integration

Twisted also comes with prebuilt implementations, including:

- A complete Web framework
- Frameworks providing facilities on top of SSH, FTP, and HTTP
- An NNTP server framework
- A user authentication system
- An instant messenger

Twisted has been the basis for a handful of other open source projects, including CVSToys, Hep, Bannerfish, Beyond 2, and DocmaServer. The users of Twisted include a number of high-profile companies like Masters of Branding, NASA, and Mailman.

Twisted programs usually use the `twisted.internet.app.Application` function. The applications created with this function are actually Python objects and can be used along with the variety of built-in tools to create and manipulative these applications Twisted comes with, just like any other Python object. The process for creating an application in Twisted normally involves creating an application object and then choosing a reactor (`twisted.inter net.reactor`), which is basically a toolkit for running Twisted on different platforms, for the application.

Reactors are the core of the event loop in Twisted, and they provide a basic interface to a number of services, including network communications, threading, and event dispatching. A reactor implements a set of interfaces, usually dependent upon which platform Twisted is playing on. After setting up an application and a rector, you can

implement Twisted network protocol parsing and handing with `twisted.internet.protocol.Protocol`.

Twisted also has `Factory` classes `(twisted.internet.protocol.Factory)` where persistent configuration is kept. The default factory classes can instantiate each protocol.

Programming in Twisted looks remarkably like Python network programming (surprise!). First you must import the reactor and protocol:

```
from twisted.internet import reactor, protocol
```

Then let's say you wanted the protocol to react to a connection:

```
class MyTwistedClass(Protocol):
def MyConnection (self):
                # do something
        self.transport.loseConnection()
```

Now set up Twisted listening on port 5555:

```
def main():
        factory = protocol.ServerFactory()
        factory.protocol = Echo
        reactor.listenTCP(5555,factory)
        reactor.run()
if _name_=='_main_':
        main()
```

Twisted itself can be found on its Sourceforge page at http://sourceforge.net/projects/twisted.

Twisted also has an active community of users and developers who can be found online at the Twisted Matrix, at http://twistedmatrix.com.

Summary

Many impressive projects have been listed and explored in this chapter. Don't be fooled, however. For each engine that I spent time researching, I had to leave out at least three others, and for every Python-based game that I played I had to miss at least two others. This is just an appetizer for what's out there waiting for the Python game developer.

Important points from this chapter:

- Python use is fairly widespread.
- It is becoming more common for games to ship with their own internal level and script editors, and Python is one of the commonalities between these tools.
- There are a number of development efforts using Python to bring the complicated task of game programming to the non-programmer.
- Most professional graphics tools include some sort of scripting interface that is Python-able.

Question and Answer

1: Q: Why didn't you mention (insert product/tool/program/ here)?

A: A: Python is so widespread and so rapidly developing that it would be impossible to list all of the games, engines, and tools that utilize it.

Exercises

1: List five industry tools that are scriptable with Python.

2: List a few of the most common uses of Python in commercial games.

3: Choose one of the engines in Section 1 of this chapter to write a two- or three screen game interface.

Part THREE: Programming with Lua

Programming with Lua and becoming comfortable with the Lua interpreter are the main focuses early on in this part of the book. Part Three also covers Lua's C API and specific industry game examples. Also included is a close-up look at LuaSDL.

Chapter 6. Programming with Lua

Language exerts hidden power, like a moon on the tides.

——Rita Mae Brown

This chapter will offer a brief introduction to the Lua language. This is a speedy overview, but the chapter does include a few common and useful examples.

Lua Executables and Debuggers

Lua can be executed in chunks written in a file or in a string by using the following function's API commands, but normally a host program executes Lua. In UNIX systems, Lua scripts can be made into executable programs by using `chmod` and placing the #! `/usr.local/bin/lua` (or whatever the Lua path is) line at the top of a Lua file. Lua files can also be executed via the Windows command line the long way (`C:\lua-5.0\bin\Lua.exe file torun.lua`), but it won't run with a double mouse click until you've set up a path or a usable development environment for Lua. For now we'll just be using Lua with the interpreter, so do not fret about it.

Lua doesn't have any built-in debugging facilities. It does, however, offer an interface with special functions and hooks that allow a programmer to construct profilers and debugging tools. These hooks are called when the interpreter enters or leaves a function or changes code. Most of these functions are new as of Lua 5.0, which is good because the older call and hook functions had the a reputation of being slow and possibly volatile. Common debug functions are listed in Table 6.1.

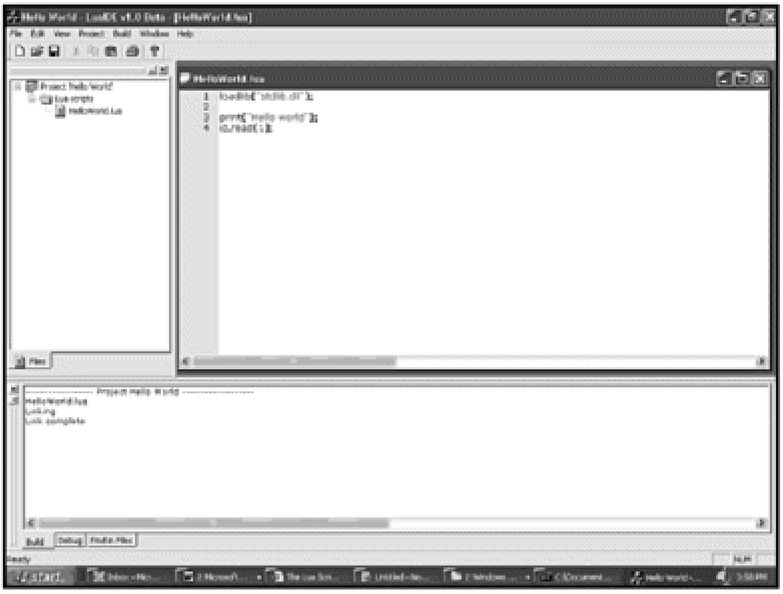
Table 6.1. Lua Debug Functions

Function	**Purpose**
`debug.gethook()`	Returns current hook settings
`debug.getinfo ()`	Returns a table with information about a function
`debug.getlocal ()`	Returns name and value of the local variable
`debug.getupvalue ()`	Returns name and value of `upvalue`
`debug.setlocal ()`	Assigns a given value to a variable
`sebug.setupvalue ()`	Assigns a given value to an `upvalue`
`debug.sethook ()`	Sets the given function of a hook
`debug.traceback ()`	Returns a string with a traceback of the call stack

For more information on the built-in debugging facilties, check out the Lua user manual, which is available from the lua.org documentation page at: http://www.lua.org/docs.html.

One interesting development tool is the LuaIDE by Tristan Rybak, which is an integrated environment for developing Lua applications (see LuaIDE in action in Figure 6.1). The environment is currently in Beta testing but is available for free for commercial or non-commercial use. Despite being a prototype, LuaIDE supports output for building and debegging messages, step-into- and stop-over-type debugger commands, breakpoints, and a callstack trace window. You can find the latest version (including source code) at Tristan Rybak's Website, at http://www.gorlice.net.pl/~rybak/luaide/.

Figure 6.1. The LuaIDE environment with a simple Lua source sample


File Edit View Project Build Window Help
HelloWorld.lua
Files
Project Hello World
HelloWorld.lua
Linking
Link complete
Build
Debug
Find in Files
Ready

Language Structure

As I mentioned, executions of Lua are broken down into units called chunks. Chunks are simply sequences of statements, and are basically equivalent to program blocks. Lua handles a chunk just like any language handles a function, so chunks can hold local variables and return values.

Chunks may be stored in a file or in a string inside the host program. When a chunk is executed, first it is precompiled into byte-code for the Lua virtual machine, and then the compiled code is executed by an interpreter for the virtual machine. Lua has no declarations, so a chunk may be as simple and short as a single statement:

```
chunk ::={single statement}
```

Or it can be big and complex:

```
Chunk ::={
        event_buffer = nil,
        last_update_ticks = 0,
        begin_time = 0,
        elapsed_ticks = 0,
        frames = 0,
        update_period = 33
        active = 1,
        screen = nil,
        background = nil,
        new_actors = {},
        actors = {},
        add_actor = function(self, a)
                assert(a)
                tinsert(self.new_actors, a)
        end
}
```

Punctuation

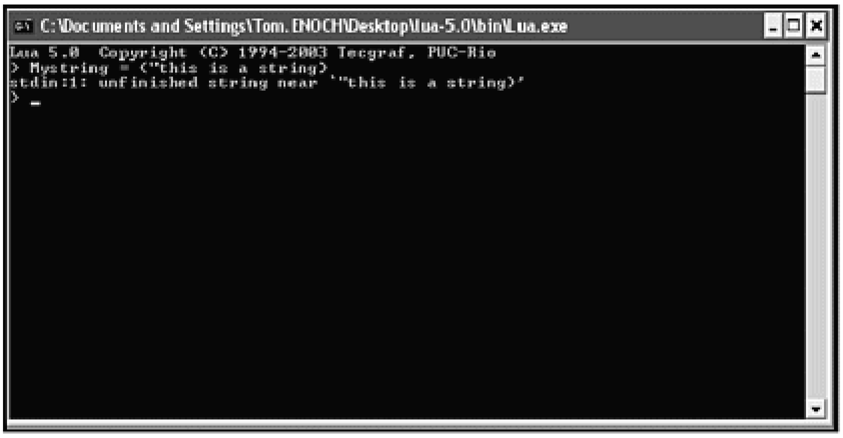
Lua uses C- and Pascal-like punctuation. This takes a bit of getting used to, especially when you're just coming from Python. While Python uses spaces and tabs to keep statements separated, Lua utilizes brackets, quotes, parentheses, squiggly lines, and other deliminators, and spaces and tabs are pretty much ignored, which can be confusing at first. A good practice is to use the interpreter often; because the interpreter expects code to be properly bracketed off, it

NOTE

I talked a bit about Pascal in earlier chapters when discussing the history of computer languages. As you may recall, Pascal is a high-level structured programming language, which forces design with a very regimented structure.

will complain immediately if you return a line of Lua that's missing something. For example, see Figure 6.2, in which our friendly interpreter reminds me that I left off the second " in the string assignment.

Figure 6.2. The Lua interpreter complains that I've left off something important



Statements in C are normally ended in a semicolon. In Lua this is optional, but you will still see it commonly done:

```
a=1
b=2
--equivalent to
a=1;
b=2;
--equivalent to
a=1;b=2;
```

Language Types

Lua is a dynamically typed language, so variables themselves do not have types; only the values of the variables have types. The basic types in Lua are shown in Table 6.2:

Variables created in Lua are visible within the blocks in which they are created and are considered global unless the area is specifically defined as local using the `local` keyword. After a code block is executed, local variables are destroyed.

Booleans

In Lua, all values different from `false` or `nil` are considered `true`. This means that only `nil` and Boolean `false` are considered `false` for the purposes of statement execution; everything else is considered `true`. As of Version 5.0, Lua has a built-in Boolean recognition of `true` and `false`.

Try running the following lines in the Lua interpreter:

Table 6.2. Built-in Data Types

Name	Data Held
`Boolean`	Either false or true
`function`	Function stored as a variable
`nil`	Value nil
`number`	Real numbers (double precision floating point)
`string`	Character string
`table`	Associative array (i.e., `dictionary` / `hash`)
`thread`	Independent threads of execution
`userdata`	C pointers stored as variable

```
x = true
print (x)
print (not x)
```

You will see that the interpreter is smart enough to know that if something is not `true`, then it must be `false`. You can use Lua to test Boolean validity by using two equal signs to represent "is equal to," like so:

```
print (0==100)
print (1 ==1)
```

Note that in Lua, `true` and `false` are not numerical values (0 and 1) like in some languages.

Functions

A really wonderful feature of Lua is that you can assign functions to variables. In fact, when you define a function in Lua, you are basically assigning the text body of the function to a given variable. Functions are declared by using the `function` keyword, with the general syntax being:

```
function name(args) does_something end
```

where `name` is the name of the new function, `args` is any arguments the function takes, `does_something` represents what the function actually does, and `end` tells Lua the function is over.

For example, here is a quick function that prints a statement to the screen:

```
function Myfunction() print("What's your function?") end
```

After creating a function, you can call it at will:

```
Myfunction()
```

You can also print the value of the function's memory address using `print`:

```
print (Myfunction)
```

When you run this last line in the interpreter, you can see that Lua notices that it's dealing with a function as well as returning its memory address.

Functions can take arguments as well, like in this example that takes an argument and assigns it to `X`:

```
function Myfunction(X) print(X) end
```

When you call this function with `Myfunction(1)`, the interpreter prints out what is assigned to `X`—in this case a 1. You could also assign the function a string with `Myfunction("hello")`. If no argument is passed to the function, Lua automatically assigns `nil` to the argument, and in the case of `Myfunction()`, the interpreter prints `nil`.

Since functions can be stored as variables in Lua, they can then be passed as arguments to other functions or they can be returned. This makes them fairly powerful creatures in Lua-land.

Nil

`Nil` values mean that a variable has no value. You can set values to `nil` to delete them:

```
x = nil
```

and you can test to see whether a variable exists by checking to see if its value is `nil`:

```
print (x==nil)
```

`Nil` is the equivalent of no value, so if a variable is assigned `nil`, it ceases to exist.

Numbers

Lua supports the standard add (`+`), subtract (`-`), multiply (`*`), and divide (`/`) operators. These can be fun to play with after firing up the lua.exe and using the `print` statement:

```
print (1+1)
print (5*5)
print (10/9)
```

If you run these lines in the interpreter, you will notice that Lua automatically brings in floating point numbers and gives you `1.11111111` as an answer to the third chunk. Lua doesn't bother with rounding off like many other languages do. All numbers in Lua are "real" numbers stored in floating point format.

You can assign numbers to variables by using the `=` sign:

```
X=100
print (x)
```

Lua also supports multiple assignments:

```
x, y = 2, 4
print (x,y)
x,y = y,x
print (x,y)
```

NOTE

The act of setting the value of a variable is called an assignment.

Lua supports the standard arithmetic relational operators, including

```
+

-

*

/

^

==

~=

<

>

<=

>=
```

These should be pretty familiar to you by now. Lua also understands logical `and`, `or`, and `not`. Logical `not` inverts a logical expression:

```
not true = false
```

while logical `and` and `or` can be used and combined to form the logical statements programmers often need:

```
true or false
x = true and y = true
```

NOTE

CAUTION

Lua does exhibit some strange behavior when ordering precedence in an equation. This behavior shows up when running through equations from left to right and right to left. Normally, Lua figures out the left side of the equals sign first, but the order in which multiple assignments are performed is actually undefined. For instance, if the same values or tables occur twice within an assignment list, then Lua may perform the equation from right to left. The order precedence may also be changed in future versions of Lua. This can be a hassle, but it simply means that you should always use separate assignment statements when possible.

An important topic for numbers and running equations is operator precedence, which is illustrated in Table 6.3.

Table 6.3. Lua Operator Precedence

Precedence	**Operator**
1.(highest)	^(exponentiation)
2.	not - (unary)
3.	* /
4.	+ -
5.	..(string concatenation)
6.(lowest)	< > <= >= ~= ==

Lua has an additional library that interfaces with the common C Math library functions. The library is available for access by Lua with a `luaopen_math` function and include a number of fun math tricks that should look familiar to C users and Math whizzes. The functions are listed in Table 6.4.

Table 6.4. Additional Math Lua Library Functions

Function	**Use**
`math.abs`	Absolute value

Table 6.4. Additional Math Lua Library Functions

Function	Use
`math.acos`	Arc cosine
`math.asin`	Arc sine
`math.atan`	Arc tangent
`math.atan2`	As atan but uses signs of the arguments to compute quadrant of the return value
`math.ceil`	Ceiling, returns smallest integer no less than given argument
`math.cos`	Cosine
`math.exp`	Exponent
`math.floor`	Returns largest integer no greater than given argument
`math.frexp`	Turns argument number into mantissa and exponent
`math.ldexp`	Returns X*(2^exp)
`math.log`	Logarithm
`math.log10`	Base-10 logarithm
`math.mod`	Splits given into integer and fraction parts
`math.pi`	Pi (3.14)
`math.pow`	Power, the base raised to exp power
`math.sin`	Sine
`math.sqrt`	Square root
`math.tan`	Tangent
`math.random`	Random number
`math.randomseed`	Seed number for random

These functions all follow a similar pattern when used. Let's say I wanted the value of pi. I'd do this:

```
MyPy = (math.pi)
print (MyPy)
```

If I needed to find the tangent of a given number, I'd do this:

```
MyTan = (math.tan(10))
print (MyTan)
```

Strings

Lua supports strings as text variable types. You can assign strings just like you would numbers, but you must be sure to include the quotes and parentheses, like so:

```
myself = ("me")
print (myself)
```

You cannot use operators like + to concatenate strings, but Lua does allow you to concatenate strings using two periods, like in the following:

```
myself = ("me")
print ("Hello to "..myself)
```

Besides double quotes, you can also set up strings using single quotes or double square brackets, as in the following:

```
--this
myself = ("me")
--is equivalent to this
myself = ('me')
--is equivalent to this
myself = ([[me]])
```

Lua supports these various methods so that you can place quotes within strings without using nasty escape sequences:

```
Mystring = ([["quote"]])
print (Mystring)
```

But Lua does support the standard C-type escape sequences when using strings. These sequences are listed in Table 6.5.

Table 6.5. Lua Escape Sequences

Sequence	**Translates to**
`\a`	System beep
`\b`	Backspace, deletes the last character typed
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\"`	Double quote

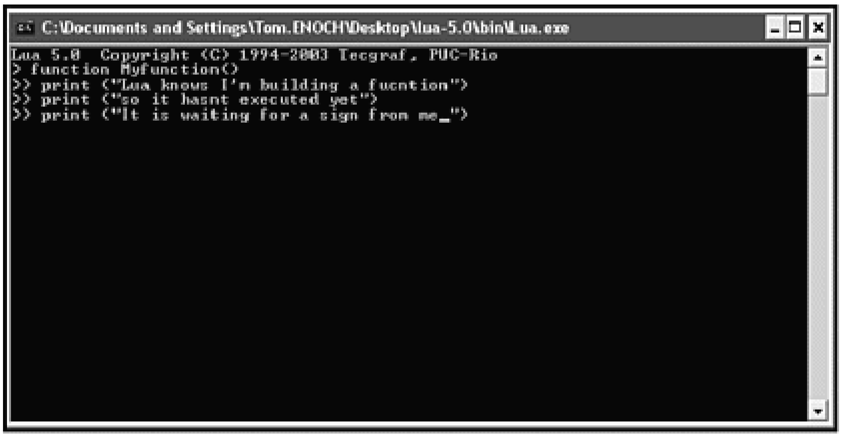
Table 6.5. Lua Escape Sequences

Sequence	Translates to
`\'`	Single quote

It is important to note that when indexing a string in Lua, the first character is at position 1 (not at 0, as with C).

Brackets have further uses when you're creating strings. For instance, they can be used to place strings on several lines of code, as shown in Figure 6.3.

Figure 6.3. Using brackets to input a string over multiple lines



Lua comes packaged with additional library string functions. These are not necessary to import Lua but are very helpful if you are working on an application with heavy string handling. These functions are listed in Table 6.6; the library is opened with the `luaopen_string` function.

Table 6.6. Lua's String-Handling Library

Function	Purpose
`string.byte ()`	Returns the internal numerical code of the character
`string.char ()`	Returns a string of given length and internal numerical codes
`string.dump ()`	Returns binary representation for a given function
`string.find ()`	Uses pattern matching to find the first match of a given string
`string.len ()`	Returns a string's length
`string.lower ()`	Returns a copy of a given string in all lowercase letters
`string.rep ()`	Returns a string concatenated to specifications given

Table 6.6. Lua's String-Handling Library

Function	Purpose
`string.sub ()`	Returns a substring of the given string
`string.upper ()`	Returns a copy of a given string in all uppercase letters
`string.format ()`	Returns a formatted version of a given string using C's `printf` style of arguments and rules
`string.gfind ()`	Used to iterate over strings to match pattern
`string.gsub ()`	Returns a copy of a given string after running given arguments over the specific string

The string library also has built-in functions for pattern matching, allowing Lua to search through long strings or tables, match up patterns, and return them (called capturing). These controls are normally preceded by modulus; they are outlined in Table 6.7.

Table 6.7. Common Pattern-Matching Controls

Symbol	Pattern
`.`	All characters
`%a`	All letters
`%c`	All control characters
`%d`	All digits
`%l`	All lowercase letters
`%p`	All punctuation characters
`%s`	All space characters
`%u`	All uppercase letters
`%w`	All alphanumeric characters
`%x`	All hexadecimal digits
`%z`	Character with representation 0

Using these functions to find patterns and matches is relatively straightforward using `string.find`. For instance, here is a Lua chunk that searches for the letter "o" in the given string:

```
MySearch = string.find('word', 'o')
print (MySearch)
```

When this chunk is run in the Lua interpreter, you are given the location of o in the string, which is the second character location, right after 'w' which is first.

Let's say that you wanted to find four-letter words that begin with s in a given string. You can use period (`.`) as a wildcard:

```
Mystring = 'Blah blah blah blah sand blah'
Mystring2 = (string.find(Mystring, 's...'))
print (Mystring2)
```

This chunk will find the word sand in the string at the 21st character location after the first four Blahs.

Tables

Tables are the main data structure in Lua. Let me repeat that, because it's important: Tables are the main data structure in Lua. Instead of lists or tuples or dictionaries, Lua utilizes tables as its primary data holder. Tables are Lua's general-purpose data type and are capable of storing groups of objects, numbers, strings, or even other tables. Tables are created using curly brackets, like so:

```
Mytable = {}
```

If you were to print out `Mytable` (using `print (MyTable)`), you would get a funny number, something like 0032bb99. This is the unique identifier and memory address that Lua has assigned to `Mytable`.

Tables are used everywhere in Lua. They are the basic building block to creating all of the important programming constructs like queues, linked lists, and arrays. Tables can also function more like hashes and dictionaries than arrays and lists. You can add hash-like objects to a table by assigning a key/value pair, like so:

```
Mytable = {Mynumber = 1, Myword = "Ikes!" }
```

You can then refer to the table with the familiar

```
print (Mytable.Mynumber)
print (Mytable.Myword)
```

Tables can also be used in an array/list-type way. You do this by creating a comma-separated list of objects when creating the table. You can then access the table like an array, using brackets and numeric references, like so:

```
Mytable = { 1,2,3,4,5,6,7,8,9,0 }
print (Mytable[1])
```

Notice, when you run this chunk in the interpreter, that the array/table starts at 1, not 0. The 0 value is actually assigned `nil`, or no value.

You can mix a dictionary-type table and array-type table together, making tables pretty versatile little buggers. Tables can also contain other tables:

```
Mytable = { table1= {a = 1, b = 2}, table2={c = 3, d = 4}}
```

Additional ways to manipulate tables are possible using the additional library functions listed in Table 6.8.

Table 6.8. Table Functions

Function	**Purpose**
`table.concat ()`	Returns concatenated tables
`table.foreach ()`	Used to execute a given function over all elements of a table
`table.foreachi ()`	Executes given function over numerical indices (only) of table
`table.getn ()`	Returns the size of the table
`table.sort ()`	Sorts tables elements in a given order
`table.insert ()`	Inserts element at a given position, shifting all other elements
`table.remove ()`	Removes element from given position, shifting elements down
`table.setn ()`	Updates the size of a table

These functions all work in a similar way. For instance, you can use `table.getn` and `table.insert` to update a table entry, like so:

```
Mytablelength = table.getn(Mytable)
--Inserts 22 into the end of the table
table.insert(Mytable, 22)
```

You can insert elements at a chosen point in the list using `table.insert`:

```
table.insert(Mytable, 10,100)
```

You can print out the contents of the table using `table.foreachi`:

```
table.foreachi(Mytable, print)
```

Even though you can treat a table as an array, keep in mind that it is still table. You can store whatever you want:

```
Mytable[5] = "Hey, a string!"
```

So, if you were printing out a dictionary version of the table

```
Mytable = {Mynumber = 1, Myword = "Ikes!" }
```

you would use the `foreach` function to print out each key/value pair:

```
table.foreach(Mytable, print)
```

The `next` function can also be used to iterate over a table. `next` takes a table and an index and gives back the next key/value pair from the table:

```
next(Mytable,"key")
```

Tables are also objects in Lua in the sense that they have state, independent identity, a life cycle, and operations that can be called upon them. The Lua programming model also has ways of implementing traditional OOP in the form of inheritance, polymorphism, classes, and late binding with tables.

People considered tables in Lua so impressive that in the latest version metatables were added as well. Every table and userdata object in Lua may now also have a metatable, which is an ordinary Lua table that further defines behavior. The commands `lua_getmetatable` and `lua_setmetatable` allow you to manipulate the metatables of a given object.

Weak tables were also added with Lua 5.0, which are tables whose elements are weak references. Unlike regular references weak references are ignored by Lua's garbage collector.. Since weak tables do not prevent garbage collection, they are useful for determining when other objects have been collected by the GC and for caching objects without impeding garbage collection.

Threads

Threads allow programs to do multiple things at once. In a multi-threading model, each task runs in a thread that is separate from other threads. There are many ways to implement multi-threading, and Lua's way is a bit unique. Lua uses a "cooperative multi-threading," using coroutines that aren't actually operating-system threads but are instead just blocks of code that can be created and run in tandem.

To create a coroutine, you first must have a function that the coroutine runs:

```
function Myfunction()
print ("do something")
coroutine.yield()
end
```

You then create a coroutine using `coroutine.create`:

```
Mythread = coroutine.create(Myfucntion)
```

Once you have established a coroutine, you can check its status with `coroutine.status`:

```
Mystatus = coroutine.status(Mythread)
print(Mystatus)
```

When run in the interpreter, this code will show that `Mythread` is suspended. To start or resume a coroutine, use `coroutine.resume`. In this example, the interpreter will print `do something`, and then `Mythread` will exit by yielding.

Yielding is key to coroutines. Coroutines must be able to yield system resources and pass control to the next thread that needs it. The `coroutine.yield` is similar to the `return` function, and it exits the current thread and frees up any resources.

If you run the `Mystatus` code a second time:

```
Mystatus = coroutine.status(Mythread)
print(Mystatus)
```

the status will show that the thread has already run by reporting `dead`.

Userdata

Userdata is used to represent C values in Lua. There are two types of userdata: full userdata and light userdata. Full userdata represents a block of memory and is considered to be an object. A light userdata represents a pointer.

Identifiers

Identifiers in Lua can be made up of letters, numbers, and underscores, but they cannot begin with a digit. Lua is case-sensitive, so the strings `HELLO` and `hello` are considered different strings. There are a handful of reserved words that Lua keeps for itself and cannot be used as identifiers; these are as follows:

```
and

break

do

else

elseif
```

```
end

false

for

function

if

in

local

nil

not

or

repeat

return

then

true

until

while
```

A standard convention in Lua is that internal variables begin with an underscore and a capital letter, like `Myvariable`.

Control Structures

Control structures in Lua are similar to those in Lua's syntactical parents C and Pascal. `if`, `while`, and `repeat` commands are very common. The traditional `if` statement looks like the following in Lua:

```
if true then block {elseif true then block} [else block] end
```

An example of an `if` statement that prints whether x is less than 10 would be:

```
x=1
if x<10 then print ("x is less than 10")end
```

You can add a second else statement in case x is greater than 10:

```
if x<10 then print ("x is less than 10")else print ("x is greater than
10")end
```

Loops

One extremely common looping statement is the `while` loop, which looks syntactically like the following:

```
while true do block end
```

A second common looping construct is the `repeat` loop:

```
repeat block until true
```

Here is a sample Lua `while` loop that prints out a series of numbers:

```
x = 1
while x<10 do
print (x)
x=x+1
end
```

The sample is just as easy to implement using `repeat`:

```
x=1
repeat
print (x)
x=x+1
until x==10
```

The `for` loop, however, is what holds a special place in the programmer's heart. Lua has two versions of the `for` loop. The first one is used with numbers:

```
for variable = var, var, var do block end
```

Like in a typical `for` loop, all three expressions aren't necessary:

```
for X=1, 10 do print(X) end
```

This loop prints X as it iterates through the loop 10 times.

The second version of `for` is used for traversing a table, and it is capable of iterating through each key/value pair of a given table:

```
for variable {, var} in explist do block end
```

An example of this version of `for` iterating over a given table is as follows:

```
Mytable = {1,2,3; word="hi, number=100000}
for key,value in Mytable do print (key,value) end
```

Included with this fun `for` is also a `pairs()` function for iterating key/value pairs:

```
for key,value in pairs(Mytable) do print (key,value) end
```

In this instance, `pairs()` will iterate only over the array type table entries in the table:

```
for index,value in ipairs(Mytable) do print (index,value) end
```

Lua uses a `return` statement to return values from a function or a Lua chunk. There is also a `break` statement that can be used to terminate the execution of a loop and skip to the next statement that follows. Both `return` and `break` must be the last statements in a given block.

Modules

Modules, packages, namespaces: all are mechanisms used by languages to organize global names and space and avoid collisions. In Lua, modules are implemented with the all-important and versatile (you guessed it) table. Identifiers become keys within tables instead of global variables. A package may look like this:

```
Mypackage = {
        function1 = function() dosomething{} end,
        function2 = function() dosomething{} end,
        function3 = function() dosomething{} end,
        function4 = function() dosomething{} end,
}
```

Then the package can be called like this:

```
call = Mypackage.function1(arguments)
```

Libraries

Lua has a set of standard libraries that provide useful and common routines. These are implemented directly through the standard API but aren't necessary to the language, and so are provided as separate C libraries. There is a basic library, a library for string manipulation, one for mathematical functions, one for system facilities and I/O, one for debugging, and one for tables. The functions are declared in `lualib.h` and must be opened with a corresponding function, like in the following examples:

```
luaopen_string
luaopen_table
```

```
luaopen_math
luaopen_io
```

A few of the libraries (math and string) were covered in the previous sections. The others will be covered here.

The Basic Library

The basic library provides much of Lua's base functionality. The commands involved are listed in Table 6.9.

The `coroutine`functions are actually part of a sublibrary of the basic library.

Input/Output Library

Input and output are handled by two file handles. These handles are stored in two global variables: `_INPUT` and `_OUTPUT`, the former for reading and the latter for writing. `_INPUT` and `_OUTPUT` are also equivalent to `_STDIN` and `_STDOUT`. The common I/O functions are listed in Table 6.10.

Table 6.10. Common Lua Input/Output Functions

Function	Purpose
`io.close ()`	Closes the given file
`io.flush ()`	Flushes over the default output file
`io.input ()`	Opens the named file in text mode and sets its handle to the default input file
`io.lines ()`	Opens the given file name in read mode and returns an iterator function that returns a new line from the file each time it is called
`io.open ()`	Opens a file in the mode specified and returns a new file handler
`io.output ()`	Opens named file in text mode and sets its handle to the default output file
`io.tmpfile ()`	Returns handle for a temporary file
`io.type ()`	Checks if object is a valid file handle
`file:close ()`	Closes file
`file:flush ()`	Saves any written data to file
`file:read ()`	Reads the file according to given formats
`file:lines ()`	Returns an integrator that returns a new line from the field each time it is called
`file:seek`	Sets and gets the file position

Table 6.10. Common Lua Input/Output Functions

Function	Purpose
`()`	
`file:write ()`	Writes the value of each of its arguments to the filehandle file

Table 6.9. Lua's Basic Function Library

Function	Purpose
`assert ()`	Issues an error when its argument is `nil`
`collectgarbage ()`	Forces a garbage collection cycle and returns the number of objects collected
`coroutine.create ()`	Creates a new coroutine
`coroutine.resume ()`	Starts or continues coroutine execution
`coroutine.status ()`	Returns status for a coroutine
`coroutine.wrap ()`	Creates a new wrapped coroutine
`coroutine.yield ()`	Suspends coroutine execution
`dofile ()`	Opens a given file and executes its contents as a Lua chunk or as precompiled chunks
`error ()`	Calls the error handler and then terminates the last protected function called
`_G`	Holds the global environment
`getfenv ()`	Returns current environment in use by a given function
`getmetatable ()`	Returns objects' `__metatable` field value or else `nil` for no metatable
`gcinfo ()`	Returns dynamic memory use and garbage collector threshold in kbytes
`ipairs ()`	Iterates over a table
`loadfile ()`	Loads a file as a Lua chunk
`loadlib ()`	Links a program to a C library
`loadstring ()`	Loads a string as a Lua chunk
`newtag ()`	Returns a new tag - equivalent to the API function `lua_newtag`
`next ()`	Allows a program to traverse all fields of a table
`pairs ()`	Iterates over tables
`pcall ()`	Calls a function in protected mode with given arguments

Table 6.9. Lua's Basic Function Library

Function	Purpose
`print ()`	Receives arguments and prints their values using the strings returned by `tostring`
`rawequal ()`	Checks to see if two values are equal
`rawget ()`	Gets the real value of an index within a table
`rawset ()`	Sets the real value of an index within a table
`require ()`	Loads a given package
`setenv ()`	Sets the environment to be used by a function
`setmetatable ()`	Sets the metatable for a given table
`tonumber ()`	Tries to convert an argument to a number
`tostring ()`	Tries to convert an argument to a string
`type ()`	Returns the type of its only argument
`tinsert ()`	Inserts an element at a given table position
`tremove ()`	Removes an element from a given table
`type ()`	Tests the type of a value
`unpack ()`	Returns all elements from a given list
`-VERSION`	Holds the current interpreter version (i.e. Lua 5.0)
`xpcall ()`	Calls a function in protected mode using `err` as the error handler

System Facilities

There are also a few system utility functions that can be included with Lua's built-in library. They are listed in Table 6.11.

Table 6.11. Lua System Facilities

Function	Purpose
`os.clock ()`	Returns an approximate CPU time, in seconds, used by the program
`os.date ()`	Returns the date and time according to given format
`os.difftime ()`	Returns the seconds between two given times
`os.execute ()`	Passes a command to be executed by the operating system. Equivalent to C's system
`os.exit ()`	Calls the C function exit to terminate a program
`os.getenv ()`	Returns the value of a given environment variable

Table 6.11. Lua System Facilities

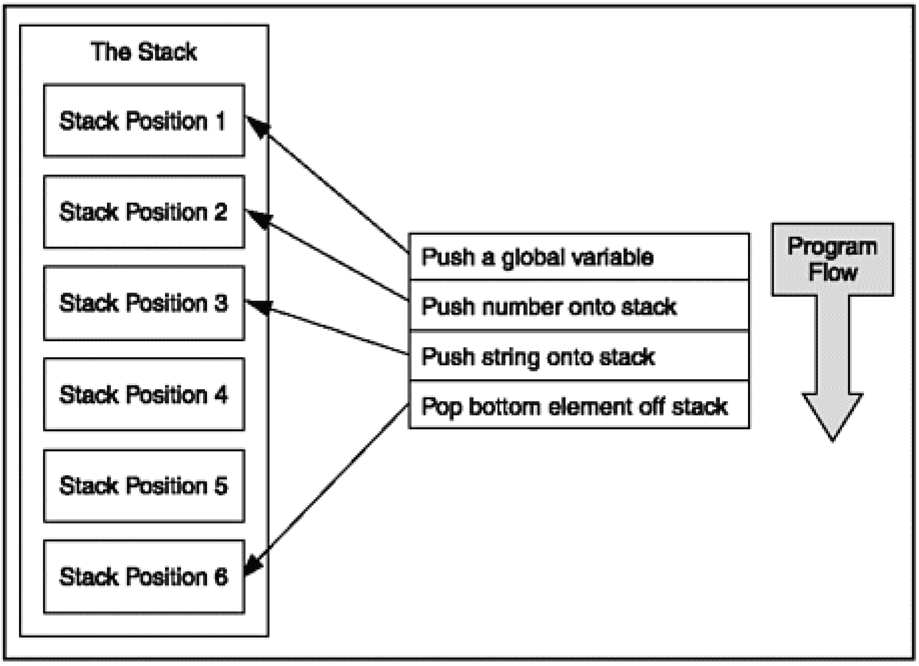
Function	**Purpose**
`os.remove ()`	Deletes a given file
`os.rename ()`	Renames a given file
`os.setlocale ()`	Used as an interface to the ANSI C `setlocale` function
`os.time ()`	Returns current time
`os.tmpname ()`	Returns a string with a filename that can be used for a temporary file

Memory, Performance, and Speed

Like most high-level languages, Lua manages memory automatically, so that you don't have to worry about allocating memory for new objects and freeing it when the objects are no longer needed. Lua manages memory automatically by running a garbage collector from time to time to collect any objects that are no longer accessible to Lua. The garbage collector picks up all of Lua's objects, including threads, tables, and so on.

Although this is not an issue when running the Lua interpreter, when calling Lua from a host, Lua's stack-in memory must be managed. Each function call in Lua needs one stack position for each argument, local variable, and temp variable, plus one position for bookkeeping. The stack should also have some 20 extra positions available. For small implementations of Lua (without, say, recursive functions), the Lua user manual suggests a stack size of 100. The default is 1,024.

Figure 6.4. A Lua script interacts with the stack




```
lua_Stat *lua_open (int stacksize);
```

To release Lua, you close its state with the stack:

```
void lua_close (lua-Stat *L);
```

This destroys all objects in a given Lua environment by calling the corresponding garbage-collection tag methods and frees all of the dynamic memory used by that state. You do not normally need to call this function because all resources are released when your program ends. However, long-running programs like Web servers or game-server

hosts may need to release states as soon as they are no longer needed so that the states don't grow too large.

When you use the Lua C API, you are responsible for controlling stack overflow. Whenever Lua calls C, it ensures that at least `LUA_MINSTACK` positions are available, so that you only have to worry about stack space when your code has loops pushing elements onto the stack. The API offers a number of functions for basic stack manipulation, including

- **void lua_settop.** Sets the stack top.
- **void lua_pushvalue.** Pushes onto the stack.
- **void lua_remove.** Removes element at given position.
- **void lua_insert.** Moves top element into given position, shifting elements on top of that position to open space.
- **void lua_replace.** Replaces a given element.

You can also query the stack with a number of functions that check the type of the given object and return strings. These functions include the following:

```
lua_type
lua_isnil
lua_isboolean
lua_isnumber
lua_isstring
lua_istable
lua_isfunction
lua_iscfunction
lua_isuserdata
lua_islightuserdata
```

`lua_equal` and `lua_rawequal` are functions for comparing two values on the stack.

To push C values onto the stack, there are a number of functions that receive C values, convert them to corresponding Lua values, and push the result onto the stack. These include:

```
lua_pushboolean
lua_pushnumber
lua_pushlstring
lua_pushstring
lua_pushnil
lua_pushcfunction
lua_pushlightuserdata
```

When chunks are called, functions like `lua-dowhile` push onto the stack any values eventually returned by the chunks. A chunk can return any number of values, and Lua checks to make sure the values fits within the stack space. But after the call, the responsibility for fitting within the stack falls back to the programmer. This means that if you need to push other elements after calling any of these functions, you should check the stack space and remove returned elements from the stack if you do not need them.

Garbage Collection

Lua uses two variables to control its garbage collection cycles. The first keeps track of how many bytes of dynamic memory Lua is using. The second variable is a threshold that, when hit, tells Lua to run the collector. These are accessible and changeable via the C API and through the `gcinfo` and `collectgarbage` functions.

Lua first counts the amount of memory it is using. If the count reaches the threshold, it runs the garbage collector. After the collection, the count is updated and the threshold is reset to twice the count value. The current count value can be retrieved with

```
lua_getccount (lu_State *L);
```

The current threshold can be retrieved with

```
lua getcthrechold (lua_State *L);
```

Each returns their values in KB. The threshold can be changed with

```
lua_setgcthreshold (lua_State *L, int newthreshold);
```

A garbage collection cycle can be forced with

```
long lua_collectgarbage(lua_State *L long limit);
```

This also returns the number of objects collected.

Garbage collector metamethods for userdata can be also set using the C API. These metemethods are called finalizers. The finalizers allow you to coordinate Lua's garbage collection with external resource management if necessary.

Speed

Lua supports coroutines as independent threads of execution. This isn't, however, a true independent multi-threaded system—it is a semi-collaborative multithreading system. That means a coroutine only suspends its execution by explicitly calling a yield routine. Lua also offers some support for multiple threads of execution via its C API, so if you have C libraries that offer kicking, then multi-threading Lua can cooperate with them.

Although garbage collection can be monitored and controlled, the main cause of low system performance is a large number of objects generated. If you are managing many objects, then the GC is an option, but it may not be always necessary.

Local variables in Lua are much quicker than global variables. This is because the locals are accessed by index. If possible, make any global variables local. Additionally, local

variables are kept on the stack and so will not affect the garbage collector (their values do not need to be collected by the garbage collector, as they are created on the stack).

`for` loops in Lua have been optimized, and also have specialized virtual machine instructions. This means that they can be faster than `while`- and `repeat`-type loops and should be used if speed is your goal.

The built-in debugger features (mainly hooks) can be used to profile Lua code and look for bottlenecks in execution time. There is also a Lua Profiler available on the lua-users.org site Wiki page, at http://lua-users.org/wiki/LuaProfiler.

When reading in files, Lua buffers the files in chunks, which is faster than reading files line by line.

Summary

Before moving on to the next chapter, you should have Lua installed on your computer and you should feel quite comfortable plugging chunks into the Lua interpreter. You should have taken a good look at Lua's structures, particularly `if/for/while`, and especially tables. You should have tried playing with a few functions from the string and math libraries. Important points from this chapter:

- Lua is normally executed by a host program or language.
- Lua code is broken up into chunks, which are similar to program blocks or single statements.
- Tables are very important in Lua.
- Lua is not designed for building huge programs. Its aim is to be useful in creating small programs or parts of a larger system.

Questions and Answers

1: Q: What about all the object-oriented features of Lua, like multiple inheritance and polymorphism?

A: A: Although Lua has worked towards OOP support, the language isn't really meant to be the huge factory-like mechanism for building giant programs. Unlike other OOP-type languages, Lua is meant to be small and flexible. Because of this, some OOP constructs may feel like hacks to the power Smalltalk developer. For this reason, I left out some of the complicated OOP features in this chapter.

2: Q: I want to know more about the Lua C API.

A: A: Start reading the next chapter!

Exercises

1: List four things tables are used to create in Lua.

2: Explain the difference between lua.exe and luac.exe.

3: Explain the concept of "chunks" in Lua.

4: Write a quick Lua program that looks for and finds white space within a text string and then deletes it (bonus points!).

Chapter 7. Getting Specific with Games in Lua

The plainest sign of wisdom is a continual cheerfulness: her state is like that of things in the regions above the moon, always clear and serene.

——Michel de Montaigne

In this chapter, you'll push the boundaries of Lua and examine game programming itself—with some help from LuaSDL. I'll also launch into the Lua C API in this chapter.

LuaSDL

LuaSDL is Simple DirectMedia Layer's binding into the Lua universe. LuaSDL has its own project page on Sourceforge, at http://sourceforge.net/projects/luasdl/. Lua users also keep a copy of the distribution on their Wiki pages, at http://luausers.org/wiki/LuaModuleLuaSdl.

You can also find a copy of LuaSDL in the Chapter 7 section of this book's CD. The LuaSDL binaries are taken from Lua users.org and precompiled and generated by Thatcher Ulrich, a programmer for Oddworld Inhabitants. Thatcher's latest LuaSDL versions can be found at his Website, at http://tulrich.com.

In Windows, you need to place the prebuilt luaSDL.dll somewhere in your path in order for SDL to function. The easiest way to do this is to drop the luaSDL.dll into your Windows system folder. Linux-platform users also need to set the path or place libluaSDL.so into their library-loading path file (which varies; usually usr/lib or usr/local/lib). Only the pre-built binaries are available at the time of this writing, and they are only available on these platforms.

NOTE

TIP

If you really want to get up-to-speed with SDL, check out the highly rated Focus on SDL, by Ernest Pazera, published by Premier Press.

Gravity: A Lua SDL Game

I first introduced SDL way back in Chapter 4, where you used it with Python to do some pretty amazing stuff. Lua's SDL bindings aren't quite as complete, and unfortunately they are also a little out-of-date. The bindings are still in beta (Version 0.3 as of this writing) and were put together using the Lua 4 interpreter (the binary module has been pre-packaged with the `toLua` tool). Because of this, all of the necessary Lua scripts are bundled with the game inside the folder (so you don't try running it with Lua 5).

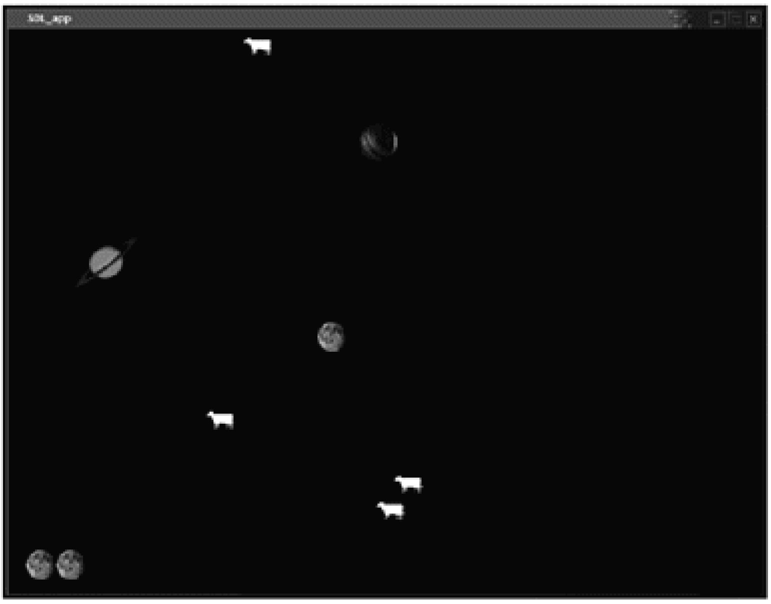
LuaSDL comes bundled with a 2D sprite game prototype called Meteor Shower. The game is written entirely in Lua and SDL by Thatcher Ulrich, who has generously given the source code to the public domain. I use this code as a base for Gravity. The entire source sample can be found in the Gravity folder in the Chapter 7 section on the CD, along with the pre-compiled DLLs necessary to use SDL and the Lua 4 interpreter.

You can launch Gravity from the command line; just navigate to the directory using the command line and type:

```
Lua Gravity.lua
```

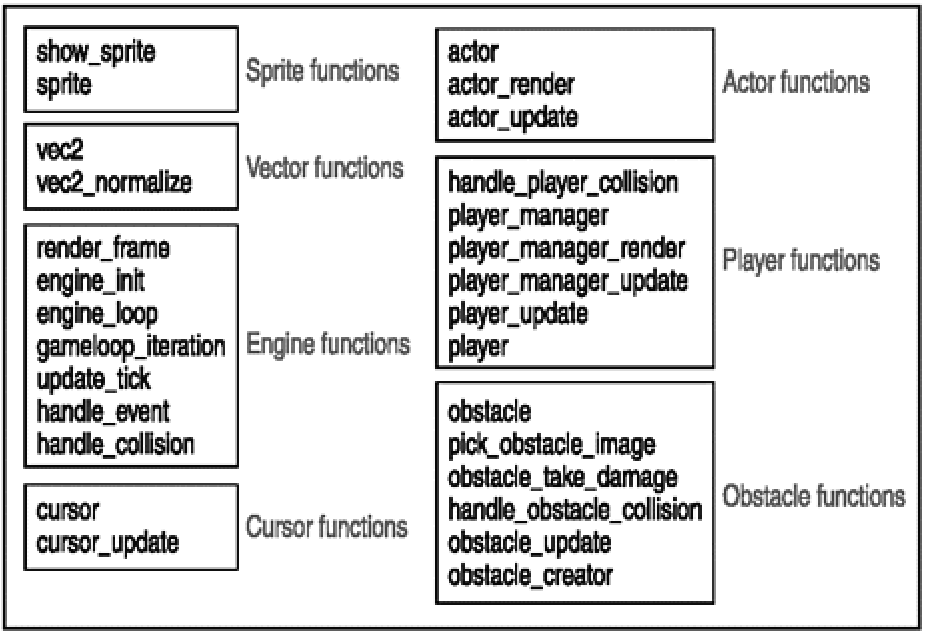
In Gravity, the player is the moon in a universe gone haywire. Planetary objects and space travelers zoom across the screen, each attracted to themselves and to the player by their given mass (see Figure 7.1). The player must avoid these objects or face destruction.

Figure 7.1. Gravity goes haywire in this LuaSDL game



A number of functions keep Gravity going. The list of functions for Gravity is shown in Figure 7.2.

Figure 7.2. The function list for Gravity



Importing SDL

Before other code can start working, the program must have access to LuaSDL. This can be achieved with only a few short lines:

```
-- Need to load the SDL module
if loadmodule then
        loadmodule("SDL")
end
```

NOTE

Lua 5 versus Lua 4

Lua 5.0 was released early in April of 2003. A number of new features came with Lua 5.0, including the following:

- Coroutines for executing many independent threads.
- Block comments for having multiple comment lines in code.
- Boolean types for true and false.
- Changes to how the API loads chunks. This is supported by new commands: `lua_load`, `luaL_loadfile`, and `luaL_loadbuffer`.
- Lightweight userdata that holds a value and not an object.
- Weak tables that assist with garbage collection.
- A faster virtual machine that is register-based.
- Standard libraries that use namespaces, although basic functions are still global.
- New methods of garbage collection, such as metamethods and other new features that make collection safe.

Along with the added features came a number of incompatibilities with previous Lua versions. Watch out for the following differences if you are a Lua 4.0 guru moving to Lua 5.0:

- Metatables have replaced the tag-method scheme.
- There are a number of changes to function calls.
- There are new reserved words (including `false` and `true`).
- Most library functions are now defined inside Lua tables.
- `lua_pushuserdata` is deprecated and has been replaced with `lau_newuserdata` and `lua_pushlightuserdata`.

Work on 5.1 has already begun, and the rumor mill has it that this next version may be available by the end of 2003.

Setting Initial Variables

You must initialize a blit surface and a start gamestate early on for this 2D game.

Blitting, as you may recall from Chapter 4, is basically rendering or drawing, and in particular is the act of redrawing an object by copying the pixels of an object onto the screen.

An SDL blit surface looks like this:

```
SDL.SDL_BlitSurface = SDL.SDL_UpperBlit;
```

The `gamestate` is a collection of state variables, assigned to a Lua table, that are initialized before the game starts to run. These are listed in Table 7.1.

Table 7.1. The `gamestate` Variables

Element	Value
`last_update_ticks`	0
`begin_time`	0
`elapsed_ticks`	0
`frames`	0
`update_period`	30
`active`	1
`new_actors`	Nested table
`actors`	Nested table
`add_actor`	Function

```
gamestate = {
        last_update_ticks = 0,
        begin_time = 0,
```

```
        elapsed_ticks = 0,
        frames = 0,
        update_period = 30,     -- interval between calls to
update_tick
        active = 1,
        new_actors = {},
        actors = {},
        add_actor = function(self, a)
                assert(a)
                tinsert(self.new_actors, a)
        end
}
```

In this table there are a number of variables set to 0 and also a few nested tables. The `update_period` is the interval in milliseconds between calls to the update tick, and `active` is a Boolean that says whether the engine is currently active or not. The `add_actor` function is also defined in this table.

The next Lua table is for a sprite cache. This cache will hold sprites that have already been loaded, so the engine won't have to try and load them on-the-fly:

```
sprite_cache = {}
```

Gravity is all about speed and velocity and, well, gravity. I envisioned flying planetary objects, each with different masses, bumping and colliding with each other in a solar system-like playing screen. To achieve this effect, I have to set gravity, how often obstacles fly onto the screen, and how many lives the player will have.

```
-- Set gravity
GRAVITY_CONSTANT = 100000
-- table of virtual masses for the different obstacle sizes
obstacle_masses = { 10, 50, 75 }
OBSTACLE_RESTITUTION = .05
-- soft speed-limit on obstacles
SPEED_TURNOVER_THRESHOLD = 4000
-- player manager actor
MOONS_PER_GAME = 3
--How often till new obstacle appears
BASE_RELEASE_PERIOD = 500
```

The three obstacles, two planets and a space cow, are illustrated in Figure 7.3. Each will use a unique bitmap image that is already included in the Gravity folder. These images are placed into a Lua table.

Figure 7.3. The three obstacles in Gravity




```
--load the bitmap obstacle images
obstacle_images = {
        { "obstacle1.bmp" },
        { "obstacle2.bmp" },
        { "obstacle3.bmp" },
}
```

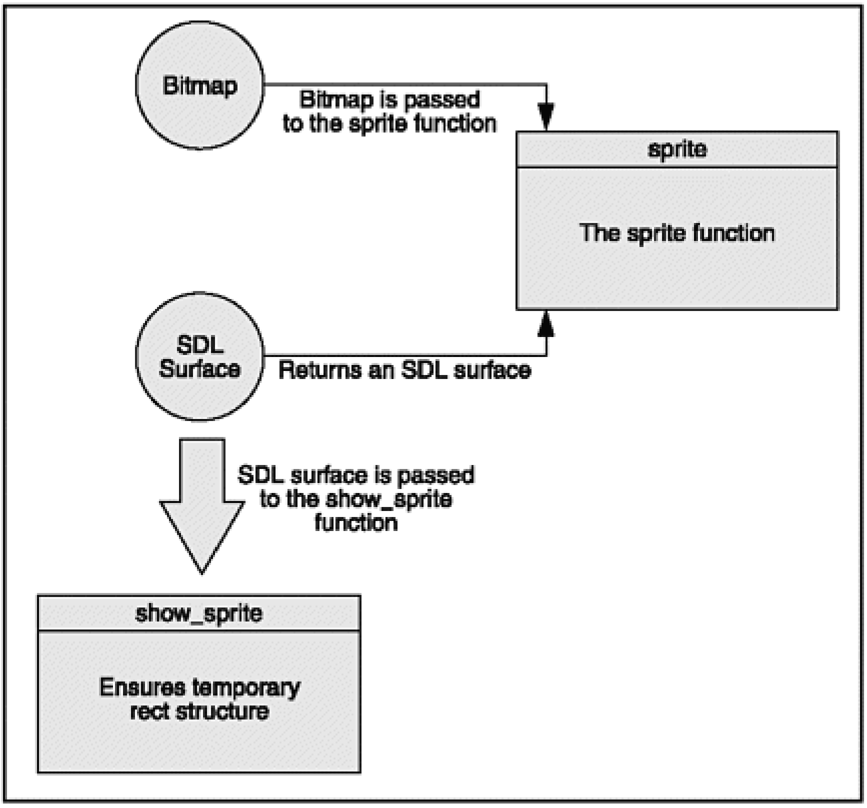
Creating Functions

Creating functions is really the meat and gravy of the endeavor. You need functions, lots of functions. Sprites, vectors, events, the game engine, and each actor (or object) within the game must be handled.

Sprite Handling

Sprite handling is the first thing to tackle (see Figure 7.4). The main sprite function will be a constructor that takes in a bitmap file and returns an SDL surface that can be blitted and used by the engine. A function that draws the new blitted SDL surface sprite onto a `rect` (`rects` are again from Chapter 4—they are the basic object for a 2D SDL game) will be part of the process as well. The main sprite function will be `sprite()`:

Figure 7.4. Sprite handling functions in Gravity




```
function sprite(file)
-- The sprite constructor. Passes in a bitmap filename and returns an
SDL_Surface
        --First check the cache
        if sprite_cache[file] then
                return sprite_cache[file]
        end
        local temp, my_sprite;
        -- Load the sprite image
        my_sprite = SDL.SDL_LoadBMP(file);
        if my_sprite == nil then
                print("Couldn't load " ..  file .. ": " ..
SDL.SDL_GetError());
                return nil
        end
        -- Set colorkey to black (for transparency)
        SDL.SDL_SetColorKey(my_sprite, SDL.bit_or(SDL.SDL_SRCCOLORKEY,
SDL.SDL_RLEACCEL), 0)
        -- Convert sprite to video SDL format
        temp = SDL.SDL_DisplayFormat(my_sprite);
        SDL.SDL_FreeSurface(my_sprite);
my_sprite = temp;
        sprite_cache[file] = my_sprite
        return my_sprite
end
```

The sprite constructor first checks to make sure that the sprite doesn't already exist in `sprite_cache`. If it does not, the constructor tries to find the given BMP image file. If the file doesn't exist, the constructor exits with an error; otherwise it goes ahead and loads the image into an SDL format (using a `temp` variable as interim), sets the

`colorkey` (another Chapter 4 concept), loads the sprite into the `sprite_cache`, and returns the sprite.

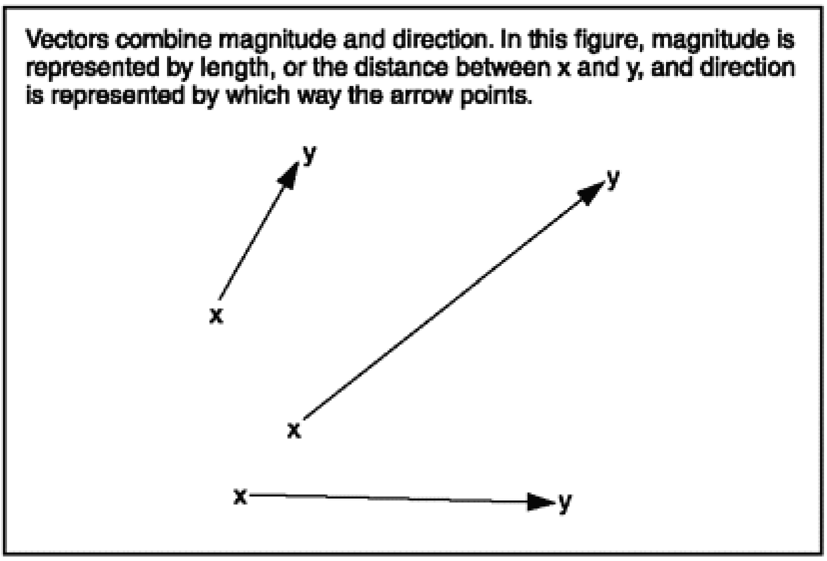
The second sprite function, `show_sprite`, is passed a sprite and draws it on the screen at the given coordinates (x,y). It uses the massively powerful `rect()` to accomplish this. Notice that in order for `show_sprite` to work, it needs all four variables:

```
function show_sprite(screen, sprite, x, y)
        -- make sure we have a temporary rect structure
        if not temp_rect then
                temp_rect = SDL.SDL_Rect_new()
        end
        temp_rect.x = x - sprite.w / 2
        temp_rect.y = y - sprite.h / 2
        temp_rect.w = sprite.w
        temp_rect.h = sprite.h
        SDL.SDL_BlitSurface(sprite, NULL, screen, temp_rect)
end
```

Vector Handling

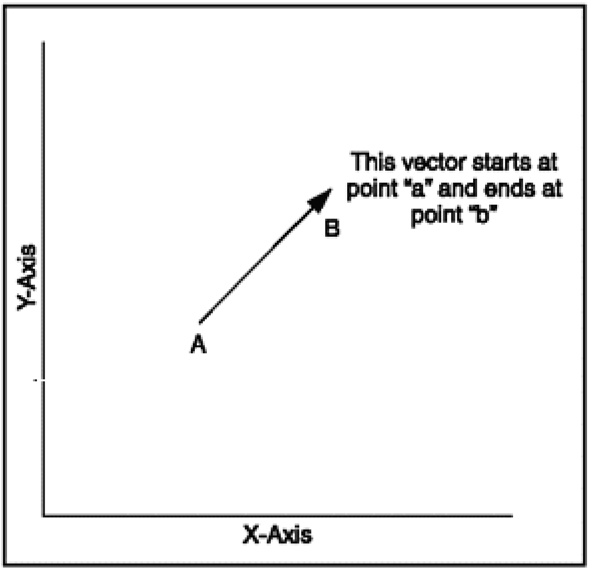
When used in game physics, vectors combine magnitude (speed) and direction (see Figure 7.5). Vectors are extremely useful, as the engine needs to know the speed and direction of the objects and actors flying around the screen. In order to do this, the `vec2` function needs to take in a table and do some math.

Figure 7.5. Vectors in physics combine magnitude and direction.



In geometry, vectors consist of a point or a location in space, a direction, and distance. The combination of direction and distance is sometimes called displacement. The `vec2` function helps to keep track of vectors using x and y coordinates, as shown in Figure 7.6. The starting coordinates are `a.x` and `a.y`, and the ending coordinates are `b.x` and `b.y`.

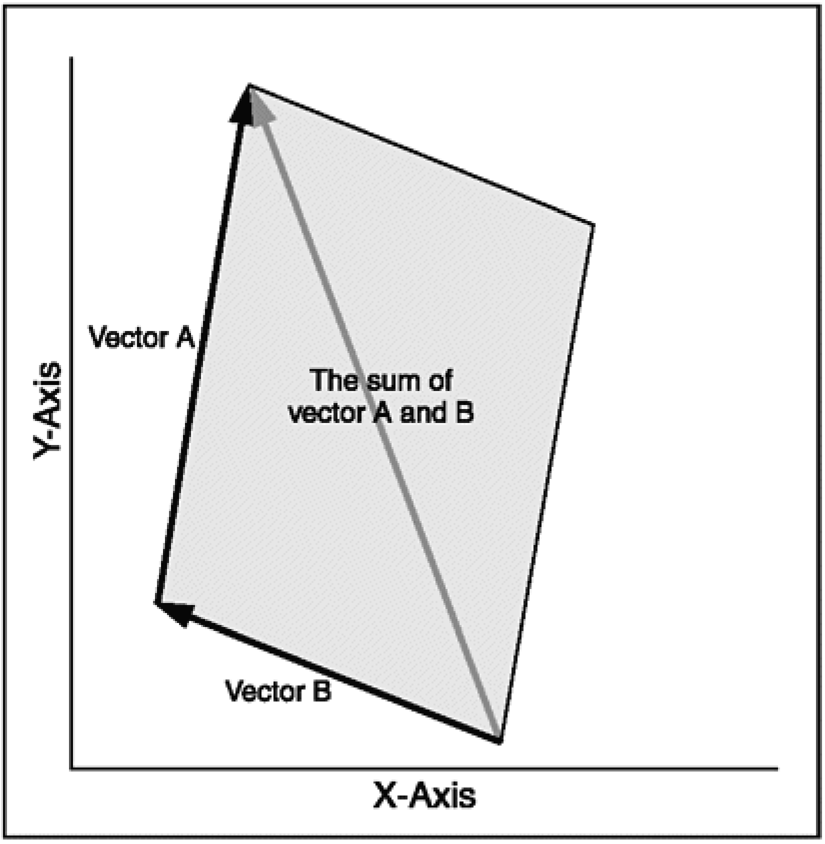
Figure 7.6. Starting and ending points of a vector



The `vec2` function has a number of methods for determining speed and direction of an actor or object using vectors. The `add`, `sub`, `mul`, and `unm` methods are used to track position in two-dimensional space by performing sector arithmetic.

The `add` method is used to do vector addition where the results of two vectors can be plotted in two-dimensional space, as shown in Figure 7.7. Vector subtraction is handled by the `sub` method, and does the opposite of vector addition by delivering the difference between two vectors.

Figure 7.7. Vector addition



You can multiply a vector by a constant to produce a second vector that travels in the same or the opposite direction but at a different speed. Multiplying vectors in math is called scalar multiplication. Scalar multipication can be really useful for collisions—say if two planets in the Gravity game collide, and they need to bounce off of each other in opposite directions.

There is also a second way of multiplying vectors that gives the angle between two vectors. This called the dot product; it is also handled by the `mul` method. Although you don't use the dot product in this game, it is a useful vector function and is sometimes used to perform lighting calculations (say, if you wanted to add a sun object that casts shadows to the game) or determine facing in 3D games.

After running through `vec2`, `vec2_normalize` finishes the vector math by dividing by the length and catching any possible close to 0 calculations that could cause errors.

```
--vec2_tag = nil
-- re-initialize the vector type when reloading
function vec2(t)
-- constructor
        if not vec2_tag then
                vec2_tag = newtag()
                Vector addition
                settagmethod(vec2_tag, "add",
                        function (a, b) return vec2{ a.x + b.x, a.y +
b.y } end
                )
```

```
                Vector subtraction
                settagmethod(vec2_tag, "sub",
                        function (a, b) return vec2{ a.x - b.x, a.y -
b.y } end
                )
                Vector multiplication
                settagmethod(vec2_tag, "mul",
                        function (a, b)
                                if tonumber(a) then
                                        return vec2{ a * b.x, a * b.y
}
                                elseif tonumber(b) then
                                        return vec2{ a.x * b, a.y * b
}
                                else
                                        -- dot product.
                                        return (a.x * b.x) + (a.y *
b.y)
                                end
                        end
                )
                settagmethod(vec2_tag, "unm",
                        function (a) return vec2{ -a.x, -a.y } end
                )
        end

        local v = {}
        if type(t) == 'table' or tag(t) == vec2_tag then
                v.x = tonumber(t[1]) or tonumber(t.x) or 0
                v.y = tonumber(t[2]) or tonumber(t.y) or 0
        else
                v.x = 0
                v.y = 0
        end
        settag(v, vec2_tag)
        v.normalize = vec2_normalize
        return v
end

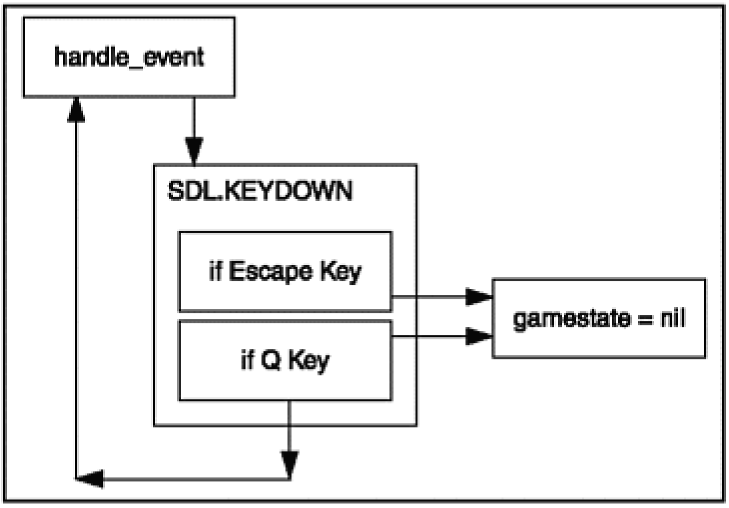
function vec2_normalize(a)
-- If a has 0 or near-zero length, sets a to an arbitrary unit vector
        local d2 = a * a
        if d2 < 0.000001 then
                -- Return arbitrary unit vector
                a.x = 1
                a.y = 0
        else
                -- divide by the length to get a unit vector
                local length = sqrt(d2)
                a.x = a.x / length
                a.y = a.y / length
        end
end
```

Event Handling

Handlers for key presses and mouse clicks are necessary for any computer game. Mouse events will be picked up by the individual actor that controls the player, but monitoring for the keyboard and windows events must also occur in case a player wants to close a

window or quit using the Escape key. This can be done fairly easily (see Figure 7.8) by using `SDL_KEYDOWN` to watch for `SDLK_q` or `SDLK_ESCAPE`.

Figure 7.8. Event handling




```
function handle_event(event)
-- called by main loop
--Checks for keypresses
-- sets gamestate to nil if player wants to quit
        if event.type == SDL.SDL_KEYDOWN then
                local   sym = event.key.keysym.sym
                if sym == SDL.SDLK_q or sym == SDL.SDLK_ESCAPE then
                        gamestate.active = nil
                end
        elseif event.type == SDL.SDL_QUIT then
                gamestate.active = nil
        end
end
```

The Engine and the Game Loop

A number of actions must happen in the engine and game loop, and these actions should correspond to a codeable function. You must have a function to remove any sprites that aren't being used and add any new ones, a function to render the screen and background, a function that keeps track of time and updates the game state, a function that does the blitting, and a function that listens for player keystrokes:

- **render_frame.** Updates and redraws.
- **engine_init.** Sets screen and video.
- **engine_loop.** Main engine loop.
- **gameloop_iteration.** Tracks time and call other functions.
- **update_tick.** Updates any game actors.
- **handle_event.** Listens for any events caused by the player.
- **handle_collision.** Handles any actor collisions.

The first step is to initialize the engine.

The engine_init function is used to set the screen width and height and the video mode and to start the game ticking, so to speak. It does all this through common-sense local variables, a few SDL calls, and calling gamestate:

```
function engine_init(argv)
        local width, height;
        local video_bpp;
        local videoflags;
        videoflags = SDL.bit_or(SDL.SDL_HWSURFACE, SDL.SDL_ANYFORMAT)
        width = 800
        height = 600
        video_bpp = 16
        -- Set video mode
        gamestate.screen = SDL.SDL_SetVideoMode(width, height,
video_bpp, videoflags);
        gamestate.background = SDL.SDL_MapRGB(gamestate.screen.format,
0, 0, 0);
        SDL.SDL_ShowCursor(0)
        -- initialize the timer/ticks
        gamestate.begin_time = SDL.SDL_GetTicks();
        gamestate.last_update_ticks = gamestate.begin_time;
end
```

Removing any actors that are no longer used and adding any new actors is handled by an update_tick function. Two Lua for loops iterate through each actor in the game. The first removes any actors that aren't active and adds any new ones:

```
for i = 1, getn(gamestate.actors) do
        if gamestate.actors[i].active then
                        -- add the actors
                        tinsert(gamestate.new_actors,
gamestate.actors[i])
        end
end
```

The former gamestate.actor table is then replaced with the new table in a quick swap:

```
gamestate.actors = gamestate.new_actors
gamestate.new_actors = {}
```

Then a second for loop calls an update for each actor in the table:

```
-- call update for each actor
        for i = 1, getn(gamestate.actors) do
                gamestate.actors[i]:update(gamestate)
        end
```

After the actors have been updated, each needs to be redrawn, as does the screen. A quick render_frame function does this work, first clearing the current screen and then redrawing each actor rect() within gamestate.actors:

```
function render_frame(screen, background)
```

```
-- When called renders a new frame.
	-- First clears the screen
	SDL.SDL_FillRect(screen, NULL, background);
	-- re-draws each actor in gamestate.actors
	for i = 1, getn(gamestate.actors) do
		gamestate.actors[i]:render(screen)
	end
	-- updates
	SDL.SDL_UpdateRect(screen, 0, 0, 0, 0)
end
```

Most of the actual game-engine work is done by this next little function, called `gameloop_iteration`. It is called each time the engine loops, and is responsible for calling all the other rendering functions and keeping track of time. First `gameloop_iteration` calls `handle_event` on any pending events in the gamestate's `event_buffer` (checking first that the buffer exists):

```
function gameloop_iteration()
-- call this to update the game state.  Runs update ticks and renders
-- according to elapsed time.
	-- if buffer doesnt exist make it so
	if gamestate.event_buffer == nil then
		gamestate.event_buffer = SDL.SDL_Event_new()
	end
	-- run handle_even on any pending events
	while SDL.SDL_PollEvent(gamestate.event_buffer) ~= 0 do
		handle_event(gamestate.event_buffer)
	end
```

`gameloop_iteration` then uses `SDL_GETTICKS()` to set the `local time` variable and compares this with the `gamestate` to see if an update needs to occur. If the engine needs to update, then `update_tick` is called and the time count is updated:

```
-- run any necessary updates
	local time = SDL.SDL_GetTicks();
	local delta_ticks = time - gamestate.last_update_ticks
	local update_count = 0
	while delta_ticks > gamestate.update_period do
		update_tick();
		delta_ticks = delta_ticks - gamestate.update_period
		gamestate.last_update_ticks =
gamestate.last_update_ticks +
gamestate.update_period
		update_count = update_count + 1
	end
```

Finally, `render_frame` has to be called to redraw any actors and the screen background if an update has occurred:

```
-- if we did any updates, then render a frame
	if update_count > 0 then
		render_frame(gamestate.screen, gamestate.background)
		gamestate.frames = gamestate.frames + 1
	end
```

```
end
```

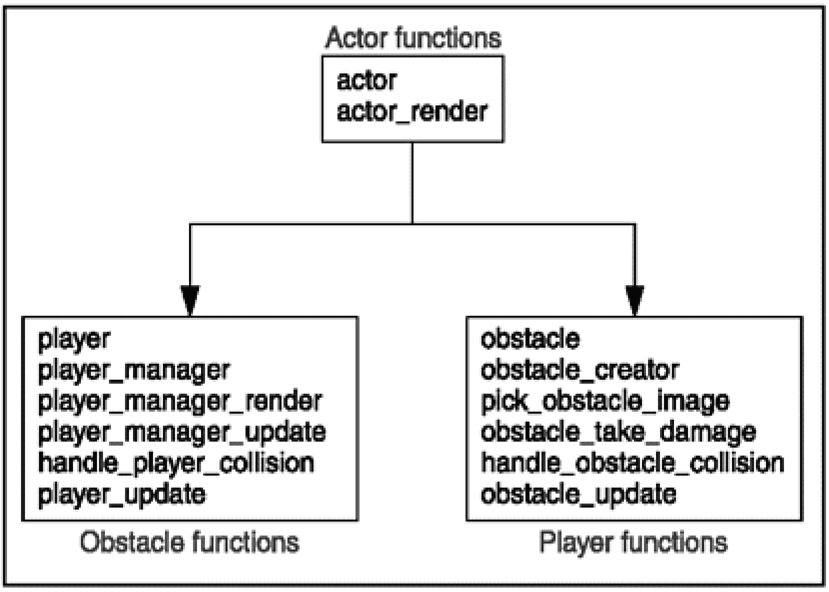
The actual engine game loop (`engine_loop`) runs while the `gamestate` is active. The `engine_loop` calls `gameloop_iteration` each time its own `while` loop fires. The `engine_loop` then cleans out the buffer. If the `gamestate` is no longer active, then `engine_loop` calls `SDL_QUIT`:

```
function engine_loop()
-- While loop calls gameloop_iteration
        while gamestate.active do
                gameloop_iteration()
        end
        -- clean up
        if event_buffer then
                SDL.SDL_Event_delete(event)
        end
        SDL.SDL_Quit();
end
```

Actors

Everyone wants to be an actor—or a computer game programmer—these days. Actors in Gravity aren't as revered or lucky as the Hollywood variety, however. They are the constructs that can be interacted with in the game, as shown in brief in Figure 7.9. These base actor functions will be used by the other objects in the game.

Figure 7.9. Actors are initialized in Gravity



Learning how to update an actor's position on the screen is the first task here, and this is where the vector functions get to stretch their legs. Velocity is multiplied by how much time has elapsed in the `gamestate` loop since the last update:

```
function actor_update(self, gs)
-- Updates than actor using vector functions
        local   dt = gamestate.update_period / 1000.0
        -- update according to velocity & time
        local   delta = self.velocity * dt
        self.position = self.position + delta
```

Since this is a 2D Asteroids-type game, objects on the screen should wrap around to the other side when they hit an edge. This effect is achieved with simple math applied to the position and the game screen (`gs.screen`) before `actor_update` ends:

```
-- wrap around at screen edge
        if self.position.x < -self.radius and self.velocity.x <= 0
then
                self.position.x = self.position.x + (gs.screen.w +
self.radius * 2)
        end
        if self.position.x > gs.screen.w + self.radius and
self.velocity.x >= 0 then
                self.position.x = self.position.x - (gs.screen.w +
self.radius * 2)
        end
        if self.position.y < -self.radius and self.velocity.y <= 0
then
                self.position.y = self.position.y + (gs.screen.h +
self.radius * 2)
        end
        if self.position.y > gs.screen.h + self.radius and
self.velocity.y >= 0 then
                self.position.y = self.position.y - (gs.screen.h +
self.radius * 2)
        end
end
```

A function that blits actors onto the screen using `show_sprite` is the next thing to create after determining the actor's position:

```
function actor_render(self, screen)
-- Blit the given actor to the given screen
        show_sprite(screen, self.sprite, self.position.x,
self.position.y)
end
```

The final curtain on actors is to build an actor constructor. The constructor will take in the sprite bitmap and keep track of position, velocity, and radius, and then return the actor in a nice, neat Lua table:

```
function actor(t)
-- actor constructor.  Pass in the name of a sprite bitmap.
        local a = {}
        -- copy elements of t
        for k,v in t do
                a[k] = v
        end
```

```
        a.type = "actor"
        a.active = 1
        a.sprite = (t[1] or t.sprite and sprite(t[1] or t.sprite)) or
nil
        a.position = vec2(t.position)
        a.velocity = vec2(t.velocity)
        a.radius = a.radius
                or (a.sprite and a.sprite.w * 0.5)
                or 0
        a.update = actor_update
        a.render = actor_render
        return a
end
```

Obstacles

The game obstacles are cows and planets. These obstacles must track a number of different things in order to make the game interesting.

- Obstacles can take damage. Some of the bigger objects will survive collisions with several smaller objects, so they need to track how much damage they can take.
- Obstacles need to know when they collide with something.
- Obstacles are drawn to each other by gravity, and so they need to keep track of other nearby obstacles.

Obstacles should also occasionally appear on the screen. They should come from offscreen at a random place, at a random speed, and travel somewhat towards the center of the screen. These object capabilities are handled with the following functions:

- **obstacle_update().** Handles gravity, movement, and collisions.
- **handle_obstacle_collision().** Called when a collision is detected.
- **obstacle_take_damage().** Damages the object.
- **pick_obstacle_image().** Chooses one of the obstacle images at random.
- **obstacle().** The obstacle constructor.
- **obstacle_creator().** Randomly places obstacles onto the screen.

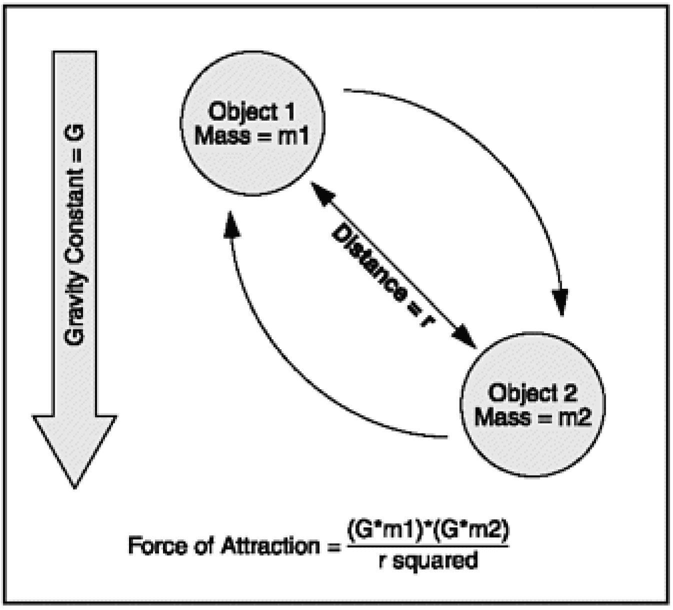
The `obstacle_update` is the first function to tackle. It watches for collisions by first updating itself and then keeping track of where the other actors are:

```
function obstacle_update(self, gs)
-- update this obstacle.  watch for collisions with other actors.
        -- move ourself
        actor_update(self, gs)
        local   dt = gamestate.update_period / 1000
        local   accel = vec2()
        -- check for the position of other actors
        for i = 1, getn(gs.actors) do
                local   a = gs.actors[i]
```

Actors with a large mass will draw other actors towards themselves. This is simulated with the `GRAVITY_CONSTANT`, the two actors' mass, and some math.

The Newtonian concept of attraction takes the mass of two objects, the distance between them, and the constant of gravity to determine how strong the attraction is between the two objects (see Figure 7.10).

Figure 7.10. Newton's law of attraction (i.e. universal gravitation)



This law is usually expressed by (G*m1)*(G*m2)/r^2, where G is the gravitational constant, m1 is the mass of the first object, m2 is the mass of the second object, and r is the distance between the two objects.

This formula is used in `obstacle_update` by taking the `GRAVITY_CONSTANT` and the mass of an object (`a.mass`) and accelerating actors towards other actors:

```
-- if the actor has mass then compute a gravitational acceleration
towards it
        if a.mass then
                local r = a.position - self.position
                local d2 = r * r
                if d2 < 100 * 100 then
                        local d = sqrt(d2)
                        if d * 2 > self.radius then
                            accel = accel + r * ((GRAVITY_CONSTANT *
a.mass) / (d2 * d))
                        end
                end
        end
```

Then `obstacle_update` needs to check for actual collisions and handle them by calling `handle_collision`. You end the function by resetting the actor's velocity:

```
        -- check for collisions, and respond
                if a and a ~= self and a.collidable then
```

```
                        local disp = a.position - self.position
                        local distance_squared = disp * disp
                        local sum_radius_squared = (a.radius +
self.radius) ^ 2
                        if distance_squared < sum_radius_squared then
                                -- we have a collision, call the
collision handler.
                                handle_collision(self, a)
                        end
                end
        end
        self.velocity = self.velocity + accel * dt
end
```

The next function, `handle_obstacle_collision`, fires when the obstacles collide. It first makes sure that the collision is between two obstacles and not between an obstacle and the player; that would be handled by a different function. It then damages the objects that collide by calling `obstacle_take_damage`:

```
function handle_obstacle_collision(a, b)
-- handles a collision between two obstacles, a and b.
        --Make sure we are handling collison between two obstacles,
otherwise exit
        if a.type == "obstacle" and b.type == "obstacle" then
           -- impulse will be along the displacement vector between
the two obstacles
           local normal = b.position - a.position
           normal:normalize()
           local relative_vel = b.velocity - a.velocity
           -- Damage the objects that collide
           local collision-energy = 0.1 * (relative_vel *
realtive_ve;) * (a.mass + b.mass)
           local split_dir = vec2{ normal.y, -normal.x }
           obstacle_take_damage(a, split_dir, -normal,
collision_energy)
           obstacle_take_damage(b, split_dir, normal,
collision_energy)
        end
end
```

The `obstacle_take_damage` is called in the event of a collision. Some objects may survive a collision, but at least one (the one with lesser mass) will be destroyed. The smallest objects (cows) will always be destroyed:

```
function obstacle_take_damage(a, split_direction, collision_normal,
collision_energy)
-- damage the obstacle; if it's damaged enough, destroy
        local split_speed = sqrt(2 * collision_energy / a.mass) * 0.35
        -- obstacle takes damage; when its damage reaches 0 it dies
        a.hitpoints = a.hitpoints - collision_energy / 2000
        if a.hitpoints > 0 then
                -- collision is not violent enough to destroy this
obstacle
                return
        end
```

```
        local new_size = a.size - 1
        if new_size < 1 then
                -- The smallest obstacle always disintegrates.
                a.active = nil
                return
        end
        -- kill a
        a.active = nil
end
```

`Pick_obstacle_image` is a short random function that will pick which object to use from the `image_table` using Lua's built-in `random`:

```
function pick_obstacle_image(size)
        local image_table = obstacle_images[size]
        -- pick one of the obstacle images at random
        return image_table[random(getn(image_table))]
end
```

The `obstacle` constructor uses the `actor` constructor as its building block. It then sets its type to `"obstacle"`, flags it as collideable, makes sure it has one of the three obstacle sizes, and then sets variables for radius, size, and speed. It also assigns the obstacle to `obstacle_update`:

```
-- constructor
-- start with a regular actor
      local a = actor(t)
      a.type = "obstacle"
      a.collidable = 1
      a.size = a.size or 3      -- make sure caller defined one of the
three sizes of obstacle
      a.sprite = sprite(pick_obstacle_image(a.size))
      a.radius = 0.5 * a.sprite.w
      a.mass = obstacle_masses[a.size]
      a.hitpoints = a.mass * a.mass
        -- implement a speed-limit on obstacles
        local   speed = sqrt(a.velocity * a.velocity)
        if speed > SPEED_TURNOVER_THRESHOLD then
                local new_speed = SPEED_TURNOVER_THRESHOLD +
sqrt(speed -
SPEED_TURNOVER_THRESHOLD)
                a.velocity = a.velocity * (new_speed / speed)
        end
        -- attach the behavior handlers
        a.update = obstacle_update
        return a
end
```

Math functions like `sqrt()` have a reputation for being slow, especially when complex math has to be calculated on-the-fly. Having to process sudden large computations can cause an otherwise fluidly running game to grind to a halt. One way to speed up `sqrt` is to cache any square root values that are used more than once. Let's say you had the following code:

```
a* sqrt(s)
b* sqrt(s)
c =  a+b
```

Instead of running the `sqrt()` function twice, run it once first and store the value:

```
square = sqrt(s)
a*square
b*square
c = a+b
```

A second trick is to do common math ahead of time and place it in a table for the program. Let's say you did a log of power of multiplication in a program; you could work out common equations first and put them in a table like Table 7.2.

Table 7.2. Common Power

Initial Value	^2	^ 3
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216

When the code needs one of these values, it gets a reference to the appropriate row and column instead of calculating on-the-fly.

The very last thing obstacles need to do is appear occasionally on the screen to harass the player. This is achieved by creating an actor that sets a countdown timer. When the timer reaches 0, the actor calls the obstacle construct, creates the obstacle on the edge of the screen, and sets it flying towards the middle somewhere. Then it starts the timer over again:

```
-- random obstacle creator
function obstacle_creator(t)
-- constructs an actor that randomly spawns a new obstacle
periodically
        a = {}
        a.active = 1
        a.type = "obstacle_creator"
        a.collidable = nil
        a.position = vec2{ 0, 0 }
        a.velocity = vec2{ 0, 0 }
        a.sprite = nil
        -- set the random timer countdown
        a.period = t.period or t[0] or 100      -- period between
spawning obstacles
        a.countdown = a.period
```

```
	a.render = function () end
	a.update =
		function (self, gs)
			self.countdown = self.countdown -
gs.update_period
			if self.countdown < 0 then
				-- timer has expired; spawn an
obstacle
				-- pick a random spot around the edge
of the screen
				local w, h = gs.screen.w, gs.screen.h
				local edge = random(w * 2 + h * 2)
				local pos
				if edge < w then
					pos = vec2{ edge, 0 }
				elseif edge < w*2 then
					pos = vec2{ edge - w, h }
				elseif edge < w*2 + h then
					pos = vec2{ 0, edge - w*2 }
				else
					pos = vec2{ w, edge - (w*2 +
h) }
				end
				-- aim at the middle of the screen
				local vel = vec2{ w/2, h/2 } - pos
				vel:normalize()
				vel = vel * (random(400) + 50)
				gs:add_actor(
					obstacle{
						size = random(3),
						position = pos,
						velocity = vel
					}
				)
				-- reset the timer
				self.countdown = self.period
			end
		end
	return a
end
```

The Player

The player is arguably the most important game piece. Much of the infrastructure the player needs (such as sprite handling and actor functions) has already been laid out. However, you still need functions to handle the following:

- Updating the player
- Player collision
- The `player` constructor

The `player_updater` function handles updating the player; it looks similar to the `object_updater` function. The `player` object is handled just like an operating system's mouse cursor. The player's position is based on the mouse position. Using `SDL_GetMouseState`, the player position is updated, and checks for any collisions are made. If there is a collision, `handle_player_collision` is called:

```
function player_update(self, gs)
-- update the player and watch for collisions
        local   dt = gamestate.update_period / 1000
        -- get the mouse position, and move the player position
towards the mouse position
        local   m = {}
        m.buttons, m.x, m.y = SDL.SDL_GetMouseState(0, 0)
        local   mpos = vec2{ m.x, m.y }
        local   delta = mpos - self.position
        local   accel =
                delta * 50      -- move towards the mouse cursor
                - self.velocity * 10    -- damping
        self.velocity = self.velocity + accel * dt
        -- move ourself
        actor_update(self, gs)
        -- check for collisions against all other actors
        for i = 1, getn(gs.actors) do
                local   a = gs.actors[i]
                -- check for collisions, and respond
                if a and a ~= self and a.collidable then
                        local disp = a.position - self.position
                        local distance_squared = disp * disp
                        local sum_radius_squared = (a.radius +
self.radius) ^ 2
                        if distance_squared < sum_radius_squared then
                                -- we have a collision
                                -- call the collision handler.
                                handle_player_collision(self, a)
                        end
                end
        end
end
```

The `handle_player_collision` also looks quite a bit like the `handle_obstacle_collision`, except it's shorter because there is no concern over damage. A collision will kill the player by setting its `active` method to `nil`:

```
function handle_player_collision(a, b)
-- handles a collision between a player, a, and some other object, b
        -- impulse will be along the displacement vector between the
two obstacle
        local normal = b.position - a.position
        normal:normalize()
        local relative_vel = b.velocity - a.velocity
        if relative_vel * normal >= 0 then
                -- don't do collision response if obstacles are moving
away from each other
                return
        end
        -- Kill the player
        a.active = nil
end
```

The `player` constructor is similar to the other constructors that have been built, except that it's smaller. The actor template is used initially, then the constructor loads the

moon.bmp as its image, sets itself as collideable, gives itself a mass (yes, the player's gravity attracts objects) and radius, and sets itself to run `player_update`.

```
function player(t)
-- constructor
        -- start with a regular actor
        local a = actor(t)
        a.type = "player"
        a.collidable = 1
        a.sprite = sprite("moon.bmp") -- or error("can't load ....")
        a.radius = 0.5 * a.sprite.w
        a.mass = 10
        -- attach the behavior handlers
        a.update = player_update
        return a
end
```

The `player` object needs a few utility functions with which to keep track of his lives and whether he's entered the game. The player cursor will have different visual states before the game starts, while playing, and after a collision, so these need to be kept track of as well. This is done with corresponding functions in the `player_manager`.

First is the `player_manager_update`. It keeps track of the player state, which is either pre-game or setup, active or playing, or deceased. If the player has died, `player_manager_update` checks to see if there are any lives left by checking the `MOONS_PER_GAME` constant. If there are, there is a short delay before the player can launch his next moon. These are all handled by a handful of Lua `if elseif then` statements:

```
function player_manager_update(self, gs)
-- keep track of game functions
        if self.state == "pre-setup" then
                -- delay, and then enter setup mode.
                self.countdown = self.countdown -
gamestate.update_period
                if self.countdown <= 0 then
                        self.state = "setup"
                        self.cursor.active = 1
                        gamestate:add_actor(self.cursor)
                end
        elseif self.state == "setup" then
                if not self.cursor.active then
                        -- player has placed the moon.  start playing.
                        self.player.active = 1
                        self.player.position = self.cursor.position
                        gamestate:add_actor(self.player)
                        -- deduct the moon that we just placed.
                        self.moons = self.moons - 1
                        self.state = "playing"
                end
        elseif self.state == "playing" then
                if not self.player.active then
                        -- player has died.
                        if self.moons <= 0 then
                                -- game is over
                                self.state = "pre-attract"
```

```
                                self.countdown = 1000
                        else
                                -- set up for next moon
                                self.state = "pre-setup"
                                self.countdown = 1000
                        end
                end
        elseif self.state == "pre-attract" then
                -- delay, and then enter attract mode
                self.countdown = self.countdown -
gamestate.update_period
                if self.countdown <= 0 then
                        self.state = "attract"
                end
        elseif self.state == "attract" then
                local m = {}
                m.buttons, m.x, m.y = SDL.SDL_GetMouseState(0, 0)
                if m.buttons > 0 then
                        -- start a new game.
                        self.state = "pre-setup"
                        self.moons = MOONS_PER_GAME

                        self.countdown = 1000
                end
        end
end
```

The function called `player_manager_render` comes in at this point to display moon sprites that show how many lives the player has left:

```
function player_manager_render(self, screen)
        if self.state == "attract" then
                show_sprite(screen, self.game_over_sprite, screen.w /
2, screen.h / 2)
        else
                -- show the moons remaining
                local sprite = self.player.sprite
                local x = sprite.w
                local y = screen.h - sprite.h
                for i = 1, self.moons do
                        show_sprite(screen, sprite, x, y)
                        x = x + sprite.w
                end
        end
end
```

The `player_manager` constructor is the last function you need to wrap up the player. Like the constructors, this function builds a Lua table that stores the variable you need, such as which player mouse curser you currently use, how many lives are left, and who to call for rendering and updating:

```
function player_manager(t)
-- constructor
        local a = {}
        for k, v in t do a[k] = v end  -- copy values from t
        a.active = 1
        a.moons = MOONS_PER_GAME
```

```
        a.state = "setup"
        a.cursor = cursor{
        }
        gamestate:add_actor(a.cursor)
        a.player = player{
                position = { gamestate.screen.w / 2,
gamestate.screen.h / 2 },
                velocity = { 0, 0 },
        }
        a.obstacle_creator.period = BASE_RELEASE_PERIOD
        a.game_over_sprite = sprite("finish.bmp")
        a.update = player_manager_update
        a.render = player_manager_render
        return a
end
```

Starting the Game

Almost finished! Only a few functions remain. The mouse cursor must be properly tracked and you need a check for mouse buttons that will start gameplay. The mouse cursor is set initially to a start.bmp graphic that lets the player choose where to position the moon when in the playing window. All of these actions are accomplished with `cursor_update` and the `cursor` constructor, and all the information is held within Lua tables:

```
function cursor_update(self, gs)
-- update the cursor.  follow the mouse.
        local   m = {}
        m.buttons, m.x, m.y = SDL.SDL_GetMouseState(0, 0)
        self.position.x = m.x
        self.position.y = m.y
        if m.buttons ~= 0 then
                -- player has clicked
                self.active = nil
        end
end

function cursor(t)
-- constructor
        -- start with a regular actor
        local a = actor(t)
        a.type = "cursor"
        a.sprite = sprite("start.bmp") -- or error("can't load ....")
        a.radius = 0.5 * a.sprite.w
        -- attach the behavior handlers
        a.update = cursor_update
        return a
end
```

Initializing the game engine is a pretty straightforward endeavor after all the work that's already been done. The `engine_init` function is called, and a slew of obstacles are in the `gamestate` with `add_actor`:

```
engine_init{}
-- Generate a bunch of obstacles
```

```
for i = 1,10 do
	gamestate:add_actor(
		obstacle{
			position = { random(gamestate.screen.w),
random(gamestate.screen.h) },
			velocity = { (random()*2 - 1) * 100,
(random()*2 - 1) * 100 }, -
- pixels/sec
			size = random(3)
		}
	)
end
```

Then create an `obstacle_creator` and a `player_manager` and let them duke it out:

```
-- create an obstracle creator
creator = obstacle_creator{}
gamestate:add_actor(creator)
-- create a player manager
gamestate:add_actor(
	player_manager{
		obstacle_creator = creator
	}
)
```

Last but not least, call the `engine_loop()`, and lo-and-behold, the game is running:

```
-- run the game
engine_loop()
```

The Lua C API

Ah, the power of C. Anything that can be done directly in Lua can also be done in the Lua C API, including manipulating variables and tables, calling functions, controlling the garbage collector, or loading Lua from strings or files.

Typically, the Lua C library is compiled into an application or run as a shared library. This is the most common way of accessing Lua in a game program. Altogether, the Lua library is very small, so it is not uncommon to find the entire source tree included with a distributed game.

NOTE

TIP

If you want to delve deeper into the C family, check out C Programming for the Absolute Beginner, by Michael Vine, or C++ Programming for the Absolute Beginner, by Dirk Henkemans and Mark Lee.

Opening Up Lua

Before calling any API function, a pointer to the Lua state must be passed as the first argument. This pointer opens up Lua. The `lua_open` command (introduced in Chapter 6) is what fires up the Lua state. All API functions need to set `lua_open` up as their very first argument.

In order to use `lua_open` in a C environment, the lua.h file must be included. The lua.h file is a C header file that defines the Lua API. However, since Lua is ANSI C, any inclusions of the Lua library must be wrapped within an `extern` C command, otherwise the compiler will mangle the names and not be able to call the commands properly. This may sound complicated, but in practice it looks like this:

```
extern "C"
{
#include <lua.h>
}
```

NOTE

Name Mangling

Compilers have a habit of modifying the names of functions and objects when compiling. This is done so that the compiler can include extra information, provide type linkage, and support function overloading. This modification is often called mangling. Particularly confusing is that each compiler has its own way of mangling names and laying out the compiled objects. This can cause problems when working with more than one language, as a second language cannot predict how a particular object or command may be mangled. Luckily, the `extern` command can be used to disable name mangling entirely.

When the Lua state machine is finished with its job, it should be closed using the `lua_close()` command. This command destroys all objects in the given Lua state via the garbage collector. Therefore, a full instance of Lua wrapped within C code looks something like this:

```
extern "C"
{
#include <lua.h>
}
lua_state *Mylua lua_open (0)
// Many lines of
// Useful Lua code that
// Do something
lua_close (MyLua)
```

More or less, every function in the Lua API deals with the Lua state or the current state of the Lua interpreter (you will often hear Lua being referred to as a "state machine" when used in this way). The Lua state keeps track of functions, globals, and any interpreter-related information. When the Lua state is closed, all the Lua objects and any dynamic memory used by the state are freed.

Whenever Lua calls C, the called function gets a virtual stack. This stack contains any arguments to the C function, is used to pass values to and from C, and will hold any values the C functions push back. Stacks can hold more than one element and are represented by an index, the top element of which can be called with `lua_gettop`:

```
Int lua_gettop (lua_State *L);
```

NOTE

```
On some platforms, you may not need to call the close state, because
resources are released normally when the program ends. Long-running
programs or daemons may need to be released occasionally.
```

Stack Commands

Lua uses a stack to pass values to and from C. Each element in this stack represents a value (`nil`, `number`, and so on) that Lua uses. The Lua API offers a number of useful commands for manipulating the stack, querying stack functions, and translating C to Lua. These commands are listed and summarized in Table 7.3.

Stack commands are normally given as arguments to the `lua_State`, a pointer to Lua (`*Lua`), and/or the appropriate index in the stack. Push functions receive a C value, convert it to a corresponding Lua value, and then push the result onto the stack.

The Lua stack is is the primary means of communication between C and Lua. There are no Lua type values in C, only functions that manipulate the stack. All values, functions, and so on are pushed onto or pulled from the stack.

Variables

Lua variables in the API do not need to be declared, and by default are considered global in scope unless specified otherwise. The variables that store Lua values are global values, local values, or table fields.

Local values can be declared anywhere within a block or chunk of Lua code. They are lexically scoped. This means the scope of variables begins at the first statement after their declaration and lasts until the end of the innermost block that includes the declaration.

Table 7.3. Lua API Stack Commands

Command	**Type**	**Purpose**
`lua_concat ();`	`void`	Concatenates the values at the top of a stack, pops them, and leaves the result at the top
`lua_equal ();`	`int`	Compares two items on the stack
lua_insert ();	void	Moves the top element to a given index
`lua_isboolean ();`	`int`	Returns 1 if the object is compatible, otherwise 0
`lua_iscfunction ();`	`int`	Returns 1 if the object is compatible, otherwise 0
`lua_isfunction ();`	`int`	Returns 1 if the object is compatible, otherwise 0
`lua_isnil ();`	`int`	Returns 1 if the object is compatible, otherwise 0
`lua_isnumber ();`	`int`	Returns 1 if the object is compatible, otherwise 0
`lua_istable ();`	`int`	Returns 1 if the object is compatible, otherwise 0
`lua_isstring ();`	`int`	Returns 1 if the object is compatible, otherwise 0
`lua_isuserdata ();`	`int`	Returns 1 if the object is compatible, otherwise 0
`lua_islightuserdata ();`	`int`	Returns 1 if the object is compatible, otherwise 0
`lua_lessthan ();`	`int`	Compares two items on the stack
`lua_pushboolean ();`	`void`	Pushes Boolean value onto the stack and returns a pointer to the Boolean
`lua_pushcfunction ();`	`void`	Pushes a C function onto the stack and returns a pointer to the function
`lua_pushfstring ();`	`void`	Pushes a formatted string onto the stack and

Table 7.3. Lua API Stack Commands

Command	Type	Purpose
		returns a pointer to the string
`lua_pushlightuserdata ();`	`void`	Pushes light user data onto the stack and returns a pointer
`lua_pushlstring ();`	`void`	Makes an internal copy of given string, pushes, and returns a pointer to the string
`lua_pushnil ();`	`void`	Pushes a `nil` value onto the stack and returns a pointer to the value
`lua_pushnumber ();`	`void`	Pushes a numeric value onto the stack and returns a pointer to the number
`lua_pushstring ();`	`void`	Pushes proper C strings onto the stack and returns a pointer to the string
`lua_pushvalue ();`	`void`	Pushes a copy of an element to a given index
`lua_pushvfstring ();`	`void`	Pushes a string onto the stack and returns a pointer to the string
`lua_rawequal ();`	`int`	Compares values for primitive equality
`lua_remove ();`	`void`	Removes element at the given index
`lua_replace ();`	`void`	Replaces given index with given element
`lua_settop ();`	`void`	Sets the stack top to a given index
`lua_State`	`struct`	Dynamic structure that holds all Lua states
`lua_totrhead();`	`int`	Converts a value on the stack into a C thread
`lua_strlen ();`	`int`	Gets a string's length
`lua_tocfunction ();`	`int`	Converts a value on the stack into a C function
`lua_tonumber ();`	`int`	Converts a Lua value at given index to a C type number. Number is a double by default
`lua_tostring ();`	`const char`	Converts a Lua value at the given index to a C type string (in C a `const *char`)
lua_touserdata ();	void	Translates userdata to a specific C type
`lua_type ();`	`int`	Returns the type of a value in a stack

All global variables exist as fields in ordinary Lua tables called environment tables or simply environments. Functions written in C and exported to Lua all share a common global environment. Each function written in Lua has its own reference to an environment, so that all global variables in that function refer to that environment table. When a function is created, it inherits the environment from the function that created it.

Userdata

Userdata is used to represent C values. Lua supports two types, full userdata and light userdata. Full userdata represents a block of memory and light user data represents a pointer. Both are considered objects.

The `lua_type` command will return `LUA_TUSERDATA` for full userdata or `LUA_TLIGHTUSERDATA` for light userdata when checking an existing userdata. New userdata can be created with the `lua_newuserdata ()` function:

```
void *lua_newuserdata (lua_stat *MyLua, size_t size);
```

This allocates a new memory block, pushes onto the stack a new userdata with the block address, and then returns the address.

Tables

The Lua API also has a few functions for manipulating metatables in objects. You create tables by calling the function `lua_newtable`. This function creates a new, empty table and then pushes it onto the stack. The function `lua_gettable`is provided for reading a value from a table that resides somewhere on the stack; when `lua_gettable`is given an index that points to the table, it will read and return the value.

Interestingly, in the Lua API, all global variables are kept within the ordinary Lua tables called environments. The initial environment that is created is called the global environment, and it can be pseudo-indexed at `LUA_GLOBALSINDEX`. Regular table operations can be used over an environment table to access and change these global values (using `lua_pushstring`, for example). The global environment of a thread can be changed using `lua_replace`.

The `lua_getfenv` and `lua_setfenv`functions are used to get and set the environment of Lua functions. First `lua_getfenv` pushes the environment table of the function on the stack at a given index, and then `lua_setfenv` pops a table from the stack and sets it as the new environment for the function at a given index.

There are a number of other useful Lua functions for dealing with tables. `Lua_getmetatable` pushes the metatable of an object on the stack, and `lua_setmetatable` sets the table on the top of a stack as a new metatable for that object and then pops the table. The `lua_load` command is used to load up Lua chunks. It automatically detects whether a chunk is text or binary, and then loads it accordingly.

```
int lua_load (lua_State *MyLua, lua_reader, void *Mydata, const char
*MyChunk);
```

The function `lua_rawget` gets the real value of a table key. To store the value into a table that resides somewhere in the stack, the key and the value are pushed by calling `lua_set table`. The `lua_rawest` function is used to set the real value of any table index. Tables can be traversed with int `lua_next`, which pops a key from the stack and

pushes a key-value pair from the table. If there are no more elements left, then `lua_next` returns a 0.

Tables are created by calling `lua_newtable`:

```
void lua_newtable (lua_State *MyLua);
```

Reading the value in a table on the stack is done by calling the `lua_gettable` command with a specific index:

```
lua_gettable (lua_State *MyLua, int specific_index);
```

Because of their universality and flexibility, tables are often used as arrays in the API.

NOTE

TIP

Some of you C buffs are probably wondering how Lua handles arrays. Lua does have functions to work with C arrays, which are treated as Lua tables and indexed by numbers. Lua basically turns Lua tables into arrays indexed by number keys. The API uses two commands to accomplish this: `lua_rawgeti`, to push the value of elements into the table at a given stack position, and `lua_rawseti`, for setting the value of elements of a table at a given stack position. The `lua_getn` command is a third function that will get the number of elements in the table/array.

Threads

Lua offers partial support for multiple threads. Since the support is pretty basic, you will often find programs that instead incorporate an existing C library offering full multi-threading.

Adding a new thread to the Lua state can be done by using the `lua_newthread`function:

```
Lua_State *lua_newthread (lua_State *L);
```

The `lua_newthread` function pushes the thread onto the stack and then returns a pointer to `lua_State` that represents this new thread. All the global objects are then shared between the different threads, but this new thread has its own independent runtime stack. Each thread also has an independent global environment table.

Manipulating an existing thread can be accomplished by using the `lua_resume` and `lua_yield` functions, which allow one to suspend or resume running threads. Lua threads can be closed using the `lua_closethread ()` function.

Calling Functions

When C and Lua are working in tandem, both C and Lua functions can be called. For C functions to work, you must do the following:

1. Register the C function with Lua.
2. Push the function to be called onto the stack.
3. Push any arguments to the function onto the stack.
4. Call the function with `lua_call`.

The `lua_call` function looks something like this:

```
int lua_call (lua_State *MyLua, int arguments, int results);
```

The `arguments` and `results` integers are the numbers of arguments and results that passed onto the stack.

If a C function needs to keep a reference to a Lua value outside of its lifespan, it must create a reference to the value. These references are stored and manipulated and released with `lua_ref, lua_getref`, and `lua_unref`.

All arguments and the function value are then popped from the stack. Lua makes sure that the returned values fit on the stack, and that the function results are pushed in direct order so that the last result is on the top. The `lua_call` function propagates any errors in this process upwards, and a special function, `lua_pcall`, is used to track error messages that flow this way.

C functions can also be used to extend Lua, a technique that is covered in Chapter 12, along with extending Ruby and Python in the same way.

Performing Actions

Lua's C API has equivalent commands to the basic library that it uses when in C API mode. These commands are listed in Table 7.4.

Table 7.4. Lua API Actions

Basic Library Function	Equivalent C API Function
`dofile ()`	`lua_dofile`
`dostring ()`	`lua_dostring`
`error ()`	`lua_error`
`newtag ()`	`lua_newtag`
`tag ()`	`lua_tag`

Table 7.4. Lua API Actions

Basic Library Function	Equivalent C API Function
`type ()`	`lua_type`

Out of all of these, `lau_dostring` is the one most likely to be encountered because it is used to perform most Lua actions. Lua can also be executed in chunks written in a file or in a string by using `lua_dofile`, `lua_dostring`, or the `lua_dobuffer` command.

When called with a `NULL` argument, `lua_dofile` executes the standard in (`stdin`) stream. Both `lua_dofile` and `lua_dobuffer` are able to execute pre-compiled Lua chunks this way. The `lua_dostring` command, however, can only execute source code.

The function `lua_dostring` calls the interpreter over a section of code contained in a string. The `lua_getglobal`, `lua_setglobal`, `lua_call`, and `lua_register` are used to interpret code files, set and manipulate global variables, call Lua functions, and make C functions accessible to Lua.

Summary

Lua's capabilities should be fairly clear at this point, and SDL has been tackled for the second time in this book. Here are a few important points before continuing to the next chapter:

- Blitting is still the key to rendering objects in SDL, whether using Python or Lua.
- Rects are still the key for blitting a sprite or object to the screen.
- The key to utilizing the C API is the stack.
- Tables in Lua are used everywhere. They make good containers for game objects and good containers for global variables in the C API.
- The most commonly found API function (after `lua_state` and `lua_open`) is `lua_dostring`.
- The Lua API functions are held within the `lua.h`header, which must be wrapped in a C `extern` command.

Questions and Answers

1: Q: I can't seem to get the Gravity.lua code to work. Is there anything else I should try?

A: A: Make sure you have the luaSDL.dll file somewhere on your system path. If you are using Windows, try this:

1. Open up a command prompt: type `cmd` or `command` from the Run option on the Start menu.

2. Navigate to the Gravity directory with the command line: use the cd command to change directories to cd MY DOCUMENTS\BOOK\CHAPTER 7\GRAVITY.

3. Type `Lua.exe Gravity.lua`

2: Q: Where can I learn more about the Lua API?

A: A: Lua-users.org Wiki pages have a few good, short API tutorials:

http://lua-users.org/wiki/

There is also an API section in the online 5.0 Lua manual:

http://www.lua.org/manual/5.0/

Exercises

1: Make a copy of the Gravity.lua source code and try playing with some of the variables to see what happens. Change the width and height of the video screen, change the number of player lives, and mess with the gravity and speed constants. What would you add or change to make the game more interesting or fun?

2: Take a look at the Meteor Shower game that comes bundled with the LuaSDL after you have a pretty good feel for Gravity to see what an even more complex Lua game looks like. Again, make some changes to the constants and variables. See if there is anything you would change to make the game more interesting or fun.

3: Take a few of the simple Lua code samples from the last chapter try to re-script them using the C API.

Chapter 8. The Lua Game Community

Daring ideas are like chessmen moved forward. They may be beaten, but they may start a winning game.

——Goethe

Of the three languages covered in this book, Lua is the most widely used in the game industry. It is already an established tool in a handful of large game shops, and it also has a history with some of the biggest games to come out on the PC. It would be folly to try to list all of the projects in which Lua has been a player (although the Lua home site has a fairly large sampling of projects). This chapter instead highlights a few key projects.

Game Engines

Game engines are tools that help program games. In Lua's case, some of these engines are open-source and some are not; some of them are aimed towards beginners and some towards advanced programmers. Some of these engines are established and complete, while others are still in raw alpha or a quiet beta. The range of engines out there is clear evidence of the language's popularity.

Arkhart

Arkhart is an original fantasy role-playing game that uses a unique engine called the Ark engine. The Ark engine and Arkhart itself are built upon Lua and SDL. Ark provides tools, a 3D client, and Lua scripting facilities to those who want to try their hand at 3D programming Lua-style. The Arkhart home page can be found at http://arkhart.nekeme.net/en/.

The Arkhart code was originally built with JavaScript and Mozilla's jslib, but it grew so large that the authors migrated to the current SDL platform. The Ark engine itself has a module for Lua scriptables, and in particular the animation files (.anm) are defined with the Lua module. The game AI is handled within its arkhart.lua file, which initializes through the Lua AI library. Areas in the game also appear to be defined by Lua files (quest.lua files to be exact).

Arkhart is published under the Gnu General Public License. The Arkhart design team is currently looking for developers and authors in both English and French.

ClanLib

ClanLib is a multi-platform game development library—perhaps one of the most popular libraries for amateur game designers today. The idea behind ClanLib is to take care of all the hard-to-develop deep functionality like sound mixing, setting up direct draw, and read-ing image files. ClanLib provides a way of dealing with sound, graphics, and networking.

ClanLib is licensed under the GNU Library General Public License and uses Lua for extending itself and for scripting. It can also be extended and scripted with Ruby, and is discussed in a bit more length in Chapter 11 of this book.

Enigma

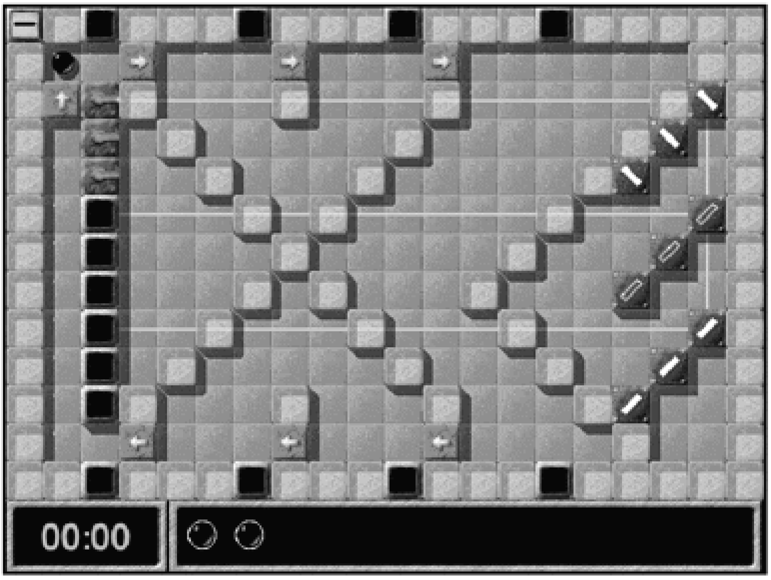
Enigma is a "nearly complete" puzzle game inspired by Atari's Oxyd and Amiga's Rock'n'Roll. Enigma is free software, with the executables and source distributed under the General Public License; it can be downloaded at the creator's Website, http://www.nongnu.org/enigma/.

Version .70 is also included on the CD in the Chapter 8 file section. Currently, executables for both Windows and Macintosh are included with the latest release, although Enigma should be playable on Posix operating systems with a bit of tweaking.

Enigma has been developed by volunteers and has a few community sites that offer levels and encouragement to new users and level designers. The game is engineered using Lua, SDL, and Oxydlib, which is a C++ library. Lua holds the distinction of being the primary language for coding different levels. Enigma is an excellent example of a cross C++/Lua project, and also a good example of how to tie the ability to script levels into a product; for these reasons, I'm going to spend some time focusing on how it works in this section.

The Enigma world is a 2D area in which the player travels in the guise of a rolling black ball (see Figure 8.1). The first step in creating a level in Enigma is to create a map of the world for the player to exist on:

Figure 8.1. The Enigma world




```
create_world(10,10)
```

This creates a 10x10 block world map. Once the map has been created, each point on the map can be accessed like a grid. The upper-left corner is always (0,0) and, in this case, the map's lower-right corner is (9,9), as you are counting from 0.

Enigma Tiles and Game Pieces

Enigma has a number of different stone tiles (prefixed by `st-`), icons (prefixed by `ix-`), items (prefixed by `it-`), floor tiles (prefixed by `fl-`), and two players (`ac-blackball` and `ac-whiteball`), although player two is currently unimplemented in the engine. These can be used to populate the world that the player travels in. Many of the standard tiles and game pieces are listed in Tables 8.1 through 8.5, although it is also possible to create your own. The Xs used in the object names indicate wildcards, where there are multiple similar tiles (for instance, there are several `st-oneway_X` tiles, a few examples being `st-oneway_white-s`, `st-oneway_black-s`, and `st-oneway_white-n`).

Table 8.1. Enigma Players

Object	Description
`ac-blackball`	Player piece
`ac-whiteball`	Second player piece (currently unimplemented)

Table 8.2. Enigma Floor Tiles

Object	Description
`fl-abyss`	Abyss floor style
`fl-bluegray`	Two color combo tile
`fl-bluegreen`	Two color combo tile
`fl-brick`	Orange brick style floor
`fl-bridge`	Bridge tile can be open or closed
`fl-dunes`	Sand tile
`fl-gradient`	Fading tile set
`fl-gray`	Gray tiles
`fl-hay`	Straw texture
`fl-himalaya`	Blue snowy tile
`fl-inverse`	Inverse of `fl-normal`
`fl-leaves`	Green forest tile
`fl-marble`	Golden stone
`fl-metal`	Metallic tiles with different rivets
`fl-normal`	Metallic tile with four corner rivets
`fl-plank`	Wood floor, planks are cross stitched
`fl-rough`	Granite-looking
`fl-sahara`	Desert tile
`fl-samba`	Stone tile segmented into four pieces
`fl-sand`	Desert tile
`fl-space`	Black with multi-colored stars
`fl-stone`	Generic stone floor
`fl-tigris`	Light marble looking floor
`fl-water`	Water floor style
`fl-wood`	Wood floor, four even strips per tile
`fl-woven`	Escher-like black and white weave

Table 8.3. Enigma Icons

Object	Description
`ic-actor`	Player icon
`ic-arrow`	Mouse pointer
`ic-bottom`	Directional arrow
`ic-down`	Directional arrow
`ic-floor`	Section of 3D grid
`ic-stone`	Picture of 3D block
`ic-top`	Directional arrow
`ic-up`	Directional arrow

Table 8.4. Enigma Items

Object	Description
`it-blackbomb`	Exploding bomb
`it-brush`	Paintbrush
`it-coin`	Money piece
`it-crack`	Crumbling segment
`it-document`	Scroll
`it-dynamite`	Stick of dynamite
`it-extralife`	Black ball (player piece)
`it-floppy`	Floppy disk
`it-hammer`	Hammer
`it-hill`	Tile bubble simulates a hill
`it-hollow`	Concave tile simulates a depression or hollow
`it-key`	Key
`it-laserX`	Different item tiles for laser items
`it-magicwand`	A magic wand
`it-magnet-off`	Magnet with no animation
`it-magnet-on`	Magnet with animation
`it-pipe`	Pipe segments
`it-seed`	Small seed bits
`it-shogund-X`	A Shogun dot, in small, medium, and large sizes
`it-spade`	Shovel
`it-spring1`	Uncompressed spring

Table 8.4. Enigma Items

Object	Description
`it-spring2`	Compressed spring
`it-surprise`	Gift package with a question mark over it
`it-sword`	Sword
`it-tinyhill`	Smaller hill
`it-tinyhollow`	Smaller hollow
`it-trigger`	Metallic trigger grate
`it-umbrella`	Umbrella
`it-wormhole`	Animated spinning wormhole
`it-yanying`	Reversed yin-yang symbol

Table 8.5. Enigma Stone Tiles

Object	Description
`st-black`	Different stones with black designs
`st-block`	Standard gray stone block
`st-bluegray`	Blue and gray fading stone
`st-bolder`	Stones with different directional arrows
`st-break`	Breaking stone animation
`st-brick`	Brick wall
`st-brownie`	Brown earthen wall
`st-coinslot`	Wall with slot for coin
`st-death`	Stone with skull and crossbones
`st-death-munch`	Skull and crossbones animation
`st-doorX`	Different stone doors
`st-fakeoxyd=blink_X`	Different blinking stones / oxyd pieces
`st-floppy1`	Stone for accepting `it-floppy` items
`st-floppy2`	Stone with `it-floppy` inserted
`st-glass`	Stone with white glass design
`st-grate1`	Closed grate
`st-grate2`	Open grate
`st-greenbrown`	Earthen green-brown stone
`st-key1`	Keyhole with no key

Table 8.5. Enigma Stone Tiles

Object	Description
`st-key2`	Keyhole with key
`st-laser-X`	Different stone tiles for lasers
`st-magic`	Stones with a keyboard look and numbers on them
`st-marble`	Generic marble stone tile
`st-metal`	Generic metal stone tile
`st-mirror-movable`	Movable mirror tile
`st-mirror-static`	Static mirror tile
`st-mirrortempl_X`	Different tiles for mirrors
`st-oneway_X`	Different stones with directional arrows
`st-oxydX`	Oxyd stone (many different game object stones)
`st-plain`	Generic plain stone wall
`st-puzzle`	Different pipe tiles
`st-rockX`	Several differently colored rock tiles
`st-rubberband`	Rubber band tile
`st-scissors`	Open scissor stone
`st-scissors-snip`	Closed scissor stone
`st-shogunX`	Several Shogun stone tiles
`st-stoneimpulse`	Impulse stone animation
`st-stoneimpulse-hollow`	Hollow impulse stone animation
`st-swap`	Broken circle
`st-switchX`	Different stoplight stones
`st-thief`	Thief stone animation
`st-timer`	Stone that triggers timed events, animated
`st-timeroff`	Triggered stone timer, no animation
`st-white`	Different white stone tiles
`st-wood`	Stone tile with wood design
`st-woven`	Escher like white weave design
`st-yellow`	Yellow stone tile
`st-yinyang1`	Yin-yang stone tile design

Creating Enigma Levels

There are a number of functions for creating levels; these are listed and described in Table 8.6.

Table 8.6. Enigma Level Design Functions

Function	Purpose	Arguments
`AddRubberBand`	Connects actors and stones that are then pulled together with given strength	Actor, object, strength, length
`create_world`	Sets base map	Width and height
`def_stone`	Defines `st-stone`	Stone name, sound
`def_floor`	Defines `fl-floor`	Floor name, friction, and mouse factor
`draw_checker-board_floor`	Draws floor alternating between two tiles	floor1, floor2, location (x,y), size (height,width), attributes
`draw_border`	Adds a border to the level	Given stone (optional: location in x,y,z coordinates and height + width)
`draw_floor`	Draws given `fl-floor`	Floor name, x and y coordinates and increments, and attributes
`draw_items`	Draws given `it-item`	Item name, x and y coordinates and increments, and attributes
`draw_stones`	Draws given `st-stone`	Stone name, x and y coordinates and increments, and attributes
`fill_floor`	Fills area with particular `st-floor`	Floor name, attributes, x and y coordinates
fill_items	Fills area with given item	Item, coordinates (x,y,z), size (height)
`fill_stones`	Fills area with given stone	Stone, coordinates(x,y,z), size (height)
`GetAttrib`	Returns current attribute value	Object, attribute name
`make_object`	Creates an object on the map, used internally by other functions	name and attributes
`set_actor`	Creates a moveable object (actor)	Name, x and y coordinates, attributes
`set_attrib`	Sets an object's attribute	The object, value and a key
`set_attribs`	Sets several attributes at once	Object, attributes
`setDefaultAttribs`	Used when placing many	Object name, attribute

Table 8.6. Enigma Level Design Functions

Function	Purpose	Arguments
	objects with same attributes	
`set_floor`	Sets given to `fl-floor`	Floor name, position (x,y), attributes
`set_item`	Sets given to `it-item`	Item name, position (x,y), attributes
`set_stone`	Sets given to `st-stone`	Floor name, position (x,y), attributes
`set_stones`	Sets given to `st-stone`, but takes multiple position arguments	Stone name, positions (x,y), attributes

There are also a few standard preset variables in Enigma, the most common being the following:

```
level_width
level_height
oxyyd_default_flavor
EAST
WEST
SOUTH
NORTH
TRUE
FALSE
```

After using `create_world` to begin an Enigma level, the next step is usually to create a frame of stones as a border around the map using the `draw_border` command. To set a border to the `st-wood`tile, do this:

```
draw_border("st-wood")
```

That's pretty simple. Now to fill the floor. By feeding `draw_checkerboard_floor` with the upper-left corner of the fill (as x and y coordinates), the map height and width (which are defined in constants already), and the two floor tiles, the floor can be filled in with alternating desert tiles:

```
fill_floor("fl-sahara","fl-sand",0,0, level_width, level_height)
```

Now that there is a filled map, you can use `set_stone` functions to create objects on the map. The `set_stone`function needs to know the type of stone and coordinates on the map and must be given a unique name (which is given as an attribute in curly braces):

```
set_stone("st-grate", 4,7, {name="My_Stone"})
```

The trick to solving a level is finding the matching onyx stones. To set these, you could also use `set_stone`:

```
set_stone("st-onyx", 1,1, {name="My_oxyd"})
```

But luckily the Enigma designers made it even easier. To save a bit of typing, use the `oxyd` command:

```
oxyd(1,1)
oxyd(2,2)
oxyd_shuffle()
```

These commands populate four game pieces, and then `oxyd_shuffle` permutes the colors on the oxyd stones within the landscape. After creating the map and the game pieces, the final step is to create the player on the map using `set_actor` using the same general conventions. The `player` attribute should always be `player=0` for the purposes of the current engine code; `5,5`are the starting x,y coordinates, and `ac-blackball` is the player piece:

```
set_actor("ac-blackball", 5,5, {player=0})
```

The Enigma source code (also included in the Enigma file folder on the CD) comes with a documents folder that includes more detailed instructions for level design, as well as many level examples (over 100) for the budding builder. The source itself is a great example of using Lua in combination with SDL.

Gime

Gime is a two-dimensional game development platform primarily used for fast prototyping. Gime uses SDL as the graphics system, and has an API that is scriptable with Lua. Gime also comes with a GUI system for creating windows and dialog boxes. Gime is written in C and is basically a glue language layer between SDL and Lua. It is currently only in prerelease (alpha) and is available at its homepage under the GNU Public License, http://www.gime.org/.

The Gime API actually has two important Lua parts: a LuaGUI library and a LuaUtil library. The LuaGUI library is capable of handling different typefaces and images. Its typface command supports both BDF and TF fonts, as well as different styles and sizes of text. Image processing is done with a wrapper to several SDL functions and allows Gime, through an image command, to create colored surfaces for text with standard opaque and alpha and colorkey settings. The GUI also supports drawing routines for filling and updating surfaces, events processing for returning information on keyboard presses and mouse movements, and a few miscellaneous functions for tracking frames, timing, and debugging.

The LuaUtil library is used for file manipulation, string manipulation, bitwise operations, and creating cache tables, which Gime uses to store value types (tables) and weak references. Gime currently requires Lua 4.0, SDL 1.2 or higher, `SDL_image`, `SDL_ttf`, freetype 2.0, and `SDL_mixer`for music.

HZ Engine

The HZ Engine is a development project by David Jeske, who wanted to re-create Herzog Zwei, a classic Sega Genesis game released in 1990 by Technosoft. Herzog Zwei was one of the first real-time strategy games and a precursor to popular titles like Command and Conquer, Total Annihilation, and Age of Empires.

Since its creation, HZ has grown into a rough platform and a nearly full real-time strategy game engine. The original version was built for Windows, but David Jeske has ported the latest to run on Linux/Xwindows. Features include

- A sprite and tile engine
- 2D hardware blit support on Win32 (which makes it a very fast engine)
- 8- or 16-bit color
- Third-person RTS-style view
- Lua scripting

HZ uses an older version of Lua (3.1) and C as its primary driver. Since many of the game's features are based on the embedded Lua, you can interactively query for information about the game using the Lua console. The backtick (`) key will bring up a Lua console while the game is being played, and you can actually script and write new code from the text console. In its current implementation, you can, by using the backtick, toggle between the game screen and the prompt that accepts Lua.

Besides being able to script events live and experiment with the Lua console while playing the game, you can completely define sprite objects using Lua. This includes everything from UI to behavior to physics.

Browsing through the source of the game (which is also available on the project Website), you can see that the engine initiates an init.lua file during the game startup. The lua.init file loads up the other necessary Lua files (using the `dofile` command from Lua's basic function library—refer to Chapter 5 for more).

Sprite initiation is one of the things Lua controls in the HZ Engine. Visually, the sprites are defined within the visrep.lua file, where you can find the code that creates the sample bases and tanks in HX. David Jenke also includes a sample sprites.lua file with examples of how to create the visual representation. A sprite that only uses one image would look like this:

```
SimpleSprite = {"image.bmp"}
```

A more complicated image with several images to indicate an animation or different traveling directions would include those images and an index:

```
ComplexSprite = {
                                        { "image1.bmp" },
                                        { "image2.bmp" },
                                        { "image3.bmp"},
                                        { "image4.bmp" };
                                IndexedBy = "CSprite"
                                }
```

The `IndexedBy` line tells the HZ Engine what object variable holds the array (table) of images. The game engine reads these values to determine which to draw (the default is the first image). You can choose one of the other images by setting the `image_frame` in the code.

The sprite logic, as well as the sprite images, are defined with Lua. The engine runs in frames, and in each frame sprites are redrawn, key presses are listened for, and sprite collisions are detected.

NOTE

CAUTION

In the existing code files, these image declarations are followed by a number of zeroes. The zeroes were for functionality that was never implemented, and they are no longer relevant or necessary, but they may cause confusion because of the obvious difference between the existing code base and the code samples.

Each sprite also has a `doTick()` method that is called at each iteration of the engine. The `doTick` method can be used to decide which image to show and set the object properties for. These properties can be anything you can dream up in Lua, but Jenke has reserved some functions in C so that the engine runs at an optimal speed. These functions are highlighted in Table 8.7.

Table 8.7. HZ Engine's C Functions for Lua Sprites

Function	Purpose
`C_obj_delete(objnum);`	Removes a sprite
`C_obj_viewFollow(objnum);`	Main camera will follow this sprite
`C_obj_getVelocity(objnum);`	Gets the velocity of a sprite
`C_obj_setVelocity(objnum,vx,vy);`	Sets the velocity of a sprite
`C_obj_getPos(objnum);`	Gets the position (x,y) of a sprite
`C_obj_setPos(objnum,x,y);`	Sets the position (x,y) of a sprite
`C_obj_setLayer(objnum,layer_number);`	Sets the graphic layer of a sprite

These functions take in `objnum` as their first parameter and x,y coordinates to follow. For instance, here's how to get an object position:

```
Co_obj_getPos(self.objnum);
```

and here's how to set the position:

```
Co_obj_setPos(self.objnum, 100, 80);
```

Having the C++ engine do the range checking and math greatly speeds up the HZ Engine. C++ is also used to handle collisions. Each sprite in HZ has a `ge_collision()` method. The point of a collision given by x and y parameters and the object that is hit are provided by a `whoIhit` parameter, which is a Lua script object.

The `keyDown` and `keyUp` event methods detect which keys are being held down. Key methods vary between platforms, making it difficult to design cross-platform, but they should suffice for game events. The `inputEvent` is used for taking in a name or typing strings from a player. More HZ documentation, the binaries, and source code can be found at David Jenke's Website and HZ project page, at http://pulp.fiction.net/~jeske/Projects/HZ.

Lixoo

Lixoo is a small, 2D, mouse-driven adventure game engine designed for conversation and character-based computer games. Lixoo consists of both the driving graphics engine and also a number of tools for users to build their games with. The main use of Lua in Lixoo is as an IDE with modules for creating rooms, characters, music, and animation.

Currently, Lixoo is under development and works only on OS X and Linux. It was originally written with ZeroForce (a small C library) but has since moved to C++. Lixoo's project page can be found on Sourceforge at http://lixoo.sourceforge.net/cgi-bin/cgilua/content.html?section=files.

The Lune Mud Server

Lune Mud is a text-based, multiuser dungeon that uses a modified Lua interpreter. Lua provides the functionality for sockets, time, and directory listings. Lune Mud runs on Linux and Win32 platforms and was written by Jason Clow.

Lune Mud is in early development but is playable. It is licensed under the GPL and can be found at Sourceforge, at http://lune.sourceforge.net.

The MADProject

An adventure-game project based on the classic Sierra Quest games, MADProject is an opensource, cross-platform, script-driven game engine, and, yes, Lua is the script that drives it. In its current iteration (as of this writing) MAD runs only on DOS and Windows, but the community is working on porting to Macintosh and Posix systems as well.

The MADProject was founded by Rick Springer. More recent development has been undertaken by project leader Nunzio Hayslip and lead programmer Javier Gonzalez, and Posix porting is being tackled by Christopher Reichenbach. MAD features include the following:

- Sprite animation
- Pathfinding
- An in-house GUI
- Music and sound effects (MIDI, WAV, and MP3)
- A Lua-based scripting interface

Windows machines must have a DLL file (alleg41.dll) placed on their path or within their systems folders in order to run the MAD sources and binaries. The engine comes with an example game called Lambazzo, whose code is the basis for the code in this section.

MAD leverages a number of other community resources besides Lua, in particular the allegro, alfont, almp3, and zlib libraries. It comes equipped with an interpreter and several utilities, all within the tools directory of the MAD source tree. Besides Lua, MAD also uses its own proprietary file format (*.mad), MAD animation files (*.anm), image files (*.img), and graphical scen files (*.scn).

The official homepage for MAD is http://mad-project.sourceforge.net. There is also a Sourceforge project page, at http://sourceforge.net/projects/mad-project/.

MAD accepts and uses full-force Lua. Lua is used to set variables and tables, perform loops, operate math, and set control structures. The latest version of MAD (of this writing) is 1.9 and is included in the Chapter 8 section on the CD.

MAD relies on a number of specific files. It searches the computer's primary archive for stdmad.lua and main.lua, the first two scripts it needs to run. Another important file is mad.cfg, which is used to determine the primary file archive and what screen size to set the display to. The mad.cfg file has the standard format of a Windows .ini file. You can also prompt mad.cfg to run in safevideo mode. Another important file is stdmad.lua, which can be hacked to alter or add custom actions and cursors to MAD.

MAD Tools

MAD files (*.mad) can be created with the MAD File Archive Manager (Mfile). Mfile can compile many game resources into a single compressed data file. Mfile is used to build MAD archives and compress the files MAD will use. The command line is used to run Mfile, and Table 8.8 lists a few of Mfile's runtime flags.

Table 8.8. Mfile Commands and Switches

Switch	Use	Example
`<`	Use a script to build a MAD file	`mfile -n MyFile.mad < MyScript.in`
`n`	Create a new archive	`mfile MyFile.mad`

Table 8.8. Mfile Commands and Switches

Switch	Use	Example
None	Open an archive	`mfile MyFile.mad`

The MAD Scene Generator (Scengen) takes as input a background image, a mask image, and a wasc image, and puts them all together to create a scene. Scengen.exe combines these three images (normally bitmap layers) into one format, a .scn format, that the MAD engine can read and use. This is done via command all on one line, naming the scene (`MyScene`) and then feeding the three bitmaps:

```
secnegen.exe MyScene.scn background.bmp mask.bmp wasc.bmp
```

After you create scenes you can view them with the MAD Scene Viewer, Sceneview. Sceneview can also be loaded with alternate resolutions by designating them on the command line. For instance, to load a scene at 640x480, do this:

```
scenview.exe MyScene.scn 640 480
```

The F10 key can be used to write bitmaps in scenview.exe into the current directory.

MAD's Animation Generator (anmgen) creates the animation file types (.anm) MAD uses. To create an animation file, you need to give anmgen the animation-creation script file (.asr) and the generated animation file (.anm) on the command line:

```
anmgen.exe MyScript.asr MyAnimation.anm
```

Animation script files have two sections separated by three percentage symbols: `%%%`. The first section lists all of the frames filenames to be used in the sub-animations:

```
walking1.bmp
walking2.bmp
walking3.bmp
%%%
```

The second section lists all of the frame filenames that are used in the sub-animations. First, the sub-animation is named, then, in parentheses, the time to display each of the frames is given:

```
walking(10)
```

A comma can be used to designate flip flags, with a 0 indicating no flipping, a 1 designating a vertical flip, a 2 designating a horizontal flip, and a 3 designating a vertical and a horizontal flip:

```
/* no flipping*/
walking(10, 0)
/*Vertical Flipping*/
walking(10, 1)
/*Horizontal Flipping*/
walking(10, 2)
/*Both Horizontal and Vertical Flipping*/
walking(10, 3)
```

After the flip frame is designated, the frame numbers are listed, separated by a space:

```
walking(10) 0 1 2
```

There is also an anmview.exe utility for viewing animation files. It loads up the animation in a viewer; then the spacebar can be pressed to play the current sub-animation. The arrow keys can be used to change the currently displayed frame.

Imgconv is an image converter that converts .bmps to MAD's image format, .img. It can convert a BMP file to a MAD image file or vice versa.

NOTE

MAD runs in 320x420 video mode with high resolution (16, 24, or 32bpp) by default.

Some video cards no longer support the classic 320x240 in 16/24/32 bit modes, and you may receive errors (something like "You need a direct x compatible video card") when trying to run MAD games. There is a `safevideo` command switch, `mad.exe -safevideo`, that you can run to get around this issue.

MAD API

MAD has an API that performs various system and engine tasks and sends information to the kernel. The functions are listed in Table 8.9.

MAD Scenes

Scenes are the background of a MAD game. Each scene is composed of three bitmaps: a 24-bit background, an 8-bit mask, and an 8-bit walk/scale. See Figure 8.2 for a sample MAD game scene.

Figure 8.2. A sample scene from MAD

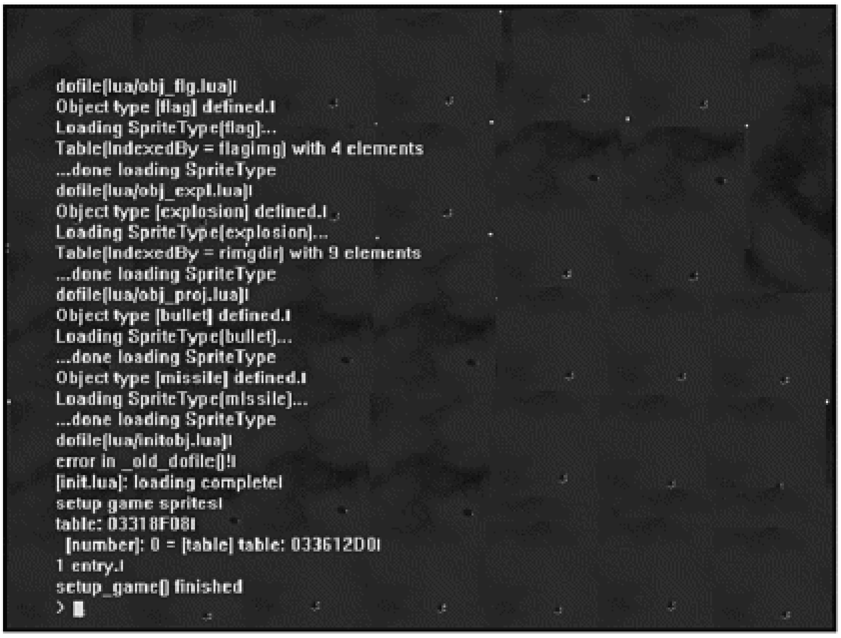


Table 8.9. MAD API Functions

Function	Purpose
`GetKey()`	Returns code of the last key pressed
`GetKeyState()`	Returns state of key constant passed to it
`GetKeyWait()`	Returns code of the last key pressed, waits if nothing has been pressed
`GetMouseBtn()`	Returns 1 if given mouse button is pressed down, 0 if it's not pressed down
`GetMouseX()`	Gets X position of the mouse pointer in pixels from top left corner of the screen
`GetMouseY()`	Gets Y position of the mouse pointer in pixels from top left corner of the screen
`GetTickCount()`	Returns time in milliseconds since MAD has started
`LoadGlobals()`	Used to load global variables or tables from a specified file
`RunScript()`	Used to have interpreter run through and add any functions or variables from a given script into the global environment
`SaveGlobals()`	Saves global variables or tables to a specified file
`SetGUIArchive()`	Sets the filename for an archive to store game files
`SetMadSpeed()`	Specifies the update speed in milliseconds; speed value of 1 is maximum speed
`SetMasterVolume()`	Used to set digital, MIDI, or MP3 volume from 0 (quiet) to 255 (loud)
`SetObjectArchive()`	Sets the filename for an archive to store game files

Table 8.9. MAD API Functions

Function	Purpose
`SetSceneArchive()`	Sets the filename for an archive to store game files
`SetScreenFX()`	Specifies an FX filter to apply to a screen after the sprites are drawn
`SetSoundArchive()`	Sets the filename for an archive to store game files

Create a scene with the following steps:

1. Assign a name (and initial memory) to the new scene.
2. Assign a particular script for the scene to run.
3. Load the actual scene file into RAM.
4. Start the scene running.

Step 1 is accomplished using the `NewScene` command:

```
My_Scene = NewScene()
```

The `SetScript` command is used to accomplish Step 2:

```
My_Scene:SetScript("My_Script.lua")
```

Loading the scene file into RAM, Step 3, is done with the `Load` command:

```
My_Scene:Load("My_Scene_File.scn")
```

And then, finally, you run the scene. In this example, running the scene causes `My_Scene_File.scn` to be drawn and `My_Script.lua` to start executing:

```
My_Scene:Run()
```

A game will likely be composed of a number of different scenes; use the `Run()` function to jump from one scene to another.

There are a couple of other scene functions for dealing with loading and unloading scenes from memory. These include

- `SetFileName.`Sets a scenes filename without loading it into memory.
- `Unload.`Frees a scene's bitmaps from memory.

- `IsLoaded()`. Checks whether or not the bitmaps for a function have been loaded into RAM.

As I mentioned, every MAD scene is composed of three bitmaps. The first is the background bitmap. The background bitmap is the actual imagery used for the background, the illustration that sets the scene; it must be 24-bit.

The Mask scene is the second bitmap, an 8-bit bitmap that is used to designate objects the player can walk behind on a background scene. Build a scene by drawing solid gray masks of the objects and then drawing a rectangle around the objects. If the rectangles of two different masks intersect, then a different shade of gray must be used so that MAD can make a designation between the two objects. These rectangles can be created in scenegen.exe by right-clicking. The scenegen.exe right-click menu also grants access to a few drawing tools, including Pencil, Paintbucket, and Undo, with a right-click. The `rectangle` command actually writes the text you'll need for the mask to a file. When drawing these rectangles, be sure to start at the top-left and move to the bottom-right; otherwise, the script will give out negative numbers.

There is some scripting involved with the mask, as well. Each object's rectangle must be defined with a `NewMaskObj()` command, so that the engine understands the size of the objects and whether other objects are drawn in front of or behind them.

The WaSc layer is the third and final bitmap layer that makes up a scene file. WaSc is short for Walk Scale, and this bitmap designates which areas of the screen the player can walk in. Areas of this mask that are painted with an index of 0 are designated as not walkable by the player.

The WaSc is also an 8-bit bitmap. In addition to designating unwalkable areas, it can also set the scale of objects drawn at given points in the scene. Depending on the background drawing, you can set the distance scale of the sprites; this is also accomplished with the index value. An index of 50 draws the objects at 50 percent, or half their original size, while an index of 100 draws the sprites at their original size.

The point in a scene that determines where a sprite is to be drawn is always the middle bottom of the sprite. This is because this is where the feet of most characters in MAD would be in a drawn sprite.

MAD Objects

Anything that a player can interact with in MAD is considered an object of some sort. The primary indicators of an object are that they move and that they are independent of their background. The steps for creating an object in MAD are as follows:

1. Allocate memory for a new object.

2. Load any animations the object will use.

3. Set the object into a scene.

4. Set any object attributes, flags, or graphic filters.

5. Show the object.

Step 1 is accomplished with the `NewObj` command:

```
My_Object = NewObj()
```

This step has to done first before any other commands can be run on an object. Objects on the move are likely to use animations of some sort, so there is a `LoadAnimation` command that will load MAD animation files (*.anm) and set the animation facing and looping:

```
My_Object:LoadAnimation("MyAnimation.anm","My_facing", 0)
```

This loads up the MAD animation, sets the animation facing to `My_facing` (which is an attribute set within the animation), and sets the looping to 0. The `SetScene` function places the object within a scene at a certain position using (x,y) coordinates:

```
My_Object:SetScene(My_Scene, 10, 200)
```

There are a handful of attributes that may or may not be necessary for a given object; `SetSize` specifies the height and width of an object and `SetSpeed` specifies the horizontal and vertical speed of an object. Object flags are also commonly used by the engine. These flags are listed in Table 8.10.

Table 8.10. MAD Object Flags

Flag	Purpose
`OBJFLAG_8WAYANIM`	Tells MAD that the object contains eight sub-animations for directional movement
`OB JFLAG_ISCHARACTER`	Sets object as a "character"
`OBJFLAG_ISEGO`	Sets object as a player-controlled character
`OBJFLAG_ISEGOAND8WAYANIM`	Sets object as both controlled character and containing eight sub-animations
`OBJFLAG_NOSCALE`	Tells MAD to not rescale `grap` to fit the scene's `wasc`
`OBJFLAG_DRAWASBKG`	Draws object as part of the background pass, before other objects
`OBJFLAG_DRAWASFRG`	Draws object as part of the foreground, after other objects are drawn

Graphic filters are set with the `SetGFXFilter` command and generally use a flag and a color (red, green, blue, or alpha) as input to create an effect when drawing an object on the screen. The following flags are defined within the stdmad.lua file:

GFXFILTER_TINT. Tints the color of an object

GFXFILTER_BLEND. Blends the object with its background

These flags work as expected. For example, let's say you want to have a few flags and manually set the speed and size of an object:

```
My_Object:SetSize(10,10)
My_Object:SetSpeed(1,1)
My_Object:SetFlags(OBJFLAG_ISCHARACTER + OBJFLAG_8WAYANIM)
My_Object:SetGFXFilter(GFXFILTER_BLEND)
```

The last step in creating a MAD object is to actually show it. All objects by default in MAD start out invisible. You use `Show` to make them appear and `Hide` to make them disappear:

```
MyObject:Show()
My_Object:Hide()
```

You can run a `kill` command to destroy or remove an object. Doing so will de-allocate memory applied to an object:

```
My_Object:Kill()
```

A number of MAD graphic functions just for objects exist; they are listed in Table 8.11.

Table 8.11. Object Graphics

Functions	**Purpose**
`GetAnimFrame()`	Returns current position of the sub-animation
`GetAnimState`	Returns 0 if animation is stopped, and a 1 if animation is running
`LoadAnimation()`	Loads a MAD animation into the object
`LoadImage()`	Loads a MAD image into the object
`PauseAnim()`	Pauses the current animation
`ResumeAnim()`	Resumes the current animation (after pausing)
`SwitchAnim()`	Changes current sub-animation and loop parameter

Path-Finding

You can set an object's position on the scene and move an object around by using the `SetPosition` command and giving MAD the (x,y) coordinates:

```
MyObject:SetPosition(1,5)
```

It isn't actually necessary to use `SetPosition` when first creating an object because `SetScene` will place the object into the scene. When an object needs to move, and move in an animated way, it is usually best to use MAD's built-in path-finding. Mad actually has a number of functions for creating mobile objects within its scene; these are listed in Table 8.12.

Table 8.12. MAD Path-Finding Functions

Function	Purpose
`GetDistance`	Calculates distance between two objects in pixels
`GetMaskDistance`	Calculates distance between an object and a mask object
`GetPosition`	Returns current (x,y) coordinates
`GetPositionChange`	Returns the change in position of the object since the last frame
`GetSpeed`	Returns speed of the object per frame
`SetPosition`	Sets object position to given coordinates
`SetPositionTL`	As above, except uses top-left positioning
`SetSpeed`	Returns horizontal and vertical speed of the object per frame
`WalkTo`	Object will walk to given coordinates. Object will move around any not walkable areas of the scene

By default, each function (except where noted) uses x and y as the coordinates within the scene. By default, MAD places an object by its middle-bottom position, the idea being that it is easier to drop a character onto a flat (2D) floor if you're using a middle-bottom position. The functions that use top-left (TL) positioning are the exceptions to this MAD rule.

Interacting with Objects

Certain object actions can be bound to script functions. This is done using a `BindAction` command and a number of object action flags. These flags correspond to cursors within the MAD GUI and are listed in Table 8.13.

Table 8.13. Object Action Flags

Flag	Cursor	Purpose
`OBJACTION_ARROW`	ARROW	Calls function when arrow cursor is used
`OBJACTION_BUSY`	BUSY	Calls function when busy cursor is used
`OBJACTION_CURITEM`	CURITEM	Calls function when the currently selected inventory item cursor is used

Table 8.13. Object Action Flags

Flag	Cursor	Purpose
`OBJACTION_CUSTOM`	CUSTOM	MAD has space for custom, programmer defined cursors
`OBJACTION_DROP`	DROP	Calls function when drop cursor is used
`OBJACTION_HELP`	HELP	Calls function when help cursor is used
`OBJACTION_EGOWALKOVER`	N/A	Calls function when EGO object walks over
`OBJACTION_LOOK`	LOOK	Calls function when the look cursor is used
`OBJACTION_TALK`	TALK	Calls function when the talk cursor is used
`OBJACTION_TARGET`	TARGET	Calls function when target cursor is used
`OBJACTION_UPDATE`	N/A	Calls function with every frame update
`OBJACTION_USE`	USE	Calls function when the use cursor is used
`OBJACTION_WALK`	WALK	Calls function when walk cursor is used

Masks

While MAD uses standard objects to handle sprites and characters that actually move around the screen, it uses `Mask` objects for immovable background pieces and decorations. `Mask` objects are considered the second type of object in MAD, but `Masks` are stationary, and their graphics are taken from the scene files and mask layer. However, the code for manipulating `Mask` objects is nearly identical to the code for manipulating objects themselves.

`Mask` objects are set up just like standard objects, but since they are based on a mask and the background layer of a scene file, not all object functions are available to them. The functions that are available, and the functions that are unique to masks, are listed in Table 8.14.

Table 8.14. `Mask` Object Functions

Function	Purpose
`BindAction()`	As object function
`GetPosition`	As object function
`Kill()`	As object function
`Hide()`	As object function
`NewMaskObj()`	Creates a new `Mask` object with given scene, (x,y) coordinates, width height, and color index
`SetFlags()`	Sets `Mask` flags
`Show()`	As object function

There is also a single `Mask`flag, `MASKOBJFLAG_NODRAW`, that will set masks to appear as part of the background but not be drawn.

Ego

The main player in MAD, otherwise known as Ego, has a number of functions with which to handle information and its display, but is otherwise just another MAD object. Ego is set with the flag `OBJFLAG_ISCHARACTER`, and must have a number of additional animations loaded with the following sub-animations:

```
eaststill
eastwalk
northstill
northwalk
southstill
southwalk
weststill
westwalk
```

If the character is also set with `OBJFLAG_8WAYANIM`, it contains the following additional walking animations:

```
nestill
newalk
nwstill
nwwalk
sestill
sewalk
swstill
swwalk
```

The MAD GUI

MAD comes with a built-in customizable GUI system that allows designers to

- Alter the mouse cursors
- Set fonts
- Create buttons and bars
- Create pop-up windows and boxes
- Customize the GUI frame or skin

These commands are outlined in Table 8.15. There are a few UI boxes that are hard-coded into the engine. These include the basic menu, the choice box, and hello world message box.

Acceptable fonts for MAD include the following:

CFF

CID-keyed Type 1 fonts

OpenType (TrueType and CFF)

SFNT-based bitmap fonts

TrueType

Type 1

Windows FNT

X11 PCF

Table 8.15. MAD GUI Functions

Function	Purpose	Notes
`AddFloatingInput()`	Creates a floating input box	
`AddFloatingText()`	Creates and returns floating text object	
`Button_BindAction()`	Binds a given function to the button	
`Button_Hide()`	Hides a button bar and all of its buttons	
`Button_LoadAnim()`	Loads an animation into the specified button	
`Button_LoadBmp()`	Loads image file into specified button	Must be MAD image format
`Button_SetFlags()`	Sets the button flags	
`Button_SetText()`	Specifies the label of a button	
`Button_Show()`	Shows a button bar and all its buttons	
`ChoiceBox()`	Displays question on screen with two choices	Hardcoded, can be positioned, stops game
`GetCursor()`	Returns `cursor_state`	
`LoadCursor()`	Specifies the mouse cursor animation	
`MenuBox()`	Displays question on screen with several choices	Hardcoded, can be positioned, stops game
`MoveFloating-`	Moves given floating	

Table 8.15. MAD GUI Functions

Function	Purpose	Notes
`InputBox()`	input box to given (x,y) coordinates	
`MoveFloatingText()`	Moves floating text object to given (x,y) coordinates	
`MsgBox()`	Window that displays messages on screen	Hardcoded, can be positioned, stops game
`NewButton()`	Creates a button inside of a button bar	
`NewButtonBar()`	Creates a bar that holds GUI buttons	
`RemoveFloating-ImnputBox()`	Removes given floating input box	
`RemoveFloatingText()`	Removes given floating text object	
`SetCursor()`	Sets selected cursor state	States are listed in Table 8.16
`SetCursorCycling()`	Enables or disables right-clicking through cursors	
`SetCursorFocus()`	Sets the cursor graphic focus point	Focus point is the (x,y) point in the mouse graphic that the screen considers "clicked"
`SetObjectUpdate-InGuiBoxd()`	Turns GUI background animations on or off	
`SetSystemFont()`	Loads a font file to be used as game text	Acceptable font formats follow
`SetTextButton-Outlines()`	Turns text button outlines on and off	

Here are steps for creating a GUI button bar:

1. Create the button bar using `MyButtonBar = NewButtonBar(10, 1, width, height)`. You must include x and y coordinates, width, and height.
2. Set any optional options, including a bar images and rgb values.
3. Add buttons to the button bar using `MyButtonBar:NewButton(width, height, ox, oy)`. The width, height, and x, y offset are required.
4. Add any optional button arguments, such as a bound function.

5. Specify the button label with `MyButtonBar:Button_SetText(MyButton, :"label")`.

6. Show the button bar on the screen with `MyButtonBar:Button_Show(MyButton)`.

The MAD mouse pointer within the GUI has a number of states that can be set. This allows the player to perform a number of different actions. There are a few built-in mouse pointer states, as well as room for a number of custom states, each of which returns a different number. These possible cursor states are outlined in Table 8.16.

Table 8.16. Possible Cursor States

State	Number Returned
`CURSOR_ARROW`	0
`CURSOR_BUSY`	1
`CURSOR_LOOK`	2
`CURSOR_WALK`	3
`CURSOR_TALK`	4
`CURSOR_USE`	5
`CURSOR_CURITEM`	6
`CURSOR_TARGET`	7
`CURSOR_DROP`	8
`CURSOR_HELP`	9
`CURSOR_CUSTOM1`	10 through 42

The keyboard is managed in a similar way to the MAD engine, with each state returning a specific number. These numbers start with `KEY_A = 1`, `KEY_B = 2`, and so on. The standard GUI skin can also be used to create custom GUI boxes, which can possess animations and custom graphics. These graphics are also referenced by number—top window border = 1, bottom window border = 2, and so on. For a complete listing of these GUI features, check out the documentation that comes with MAD and is also included on this book's CD.

MAD Sounds

MAD can load and play .wav, .voc, .mid, and .mp3 files for sound effects and music. To play sounds or effects in MAD, follow these steps:

1. Initialize the sound object.

2. Load the sound file.

3. Play the sound file.

4. Delete the sound file when it's done.

Step 1 is accomplished with a simple declaration, `NewSound()`, which loads a new sound structure into memory:

```
My_Sound = NewSound()
```

After the sound is in memory, you can use `LoadWave` or `LoadMp3` to load a particular sound file:

```
My_Sound:LoadWav("My_Wav_File.wav")
```

Then play the sound using `Play`:

```
My_Sound:Play(0)
```

`Play`takes input on how many times to loop the sound, in this case a big 0.

Finally, delete the sound using `DeleteSound()`:

```
DeleteSound(My_Sound)
```

Playing a music loop is an almost identical process. The `NewMusic` command is used instead of the `NewSound` command, and .mid files replace .wav files, although MP3s can also be used with `NewMusic`:

```
My_Sound = NewMusic()
My_Sound:LoadMidi("My_Wav_File.mid")
My_Sound:Play(0)
DeleteSound(My_Sound)
```

Items and Spells

The MAD engine handles spells that the player casts and the items that he uses in a nearly identical way. Each is associated with an ID number, and the actions performed by spells or items are left for the programmer to script. The spells and inventory items are handled the same way. The `ShowInventory` and `ShowSpells` commands take in the following parameters:

- x and y coordinates (`x,y`)
- Back window texture (`MyTexture.img`)
- Total size of the window (`window_width, window_height`)
- x and y coordinates for the item box, where all inventory items are drawn in (`itembox_width, itembox_height`)

- Any item box offset `(itembox_ox, itembox_oy)`
- Icon size (`itemicon_width, itemicon_height`)

The Inventory window in game can be toggled on and off using the `HideInventory` command. Items within the Inventory box can be either bitmaps or animations. Inventory items are added to the window using `AddItemToInv`. `AddItemToInv` also takes in a number of parameters:

- MyItem.img, which is the bitmap filename to use.
- MyItem.anm, which is the animation filename to use.
- Item Name is the name of the item.
- Weight is how much the unit weighs in game units.
- Quantity is how many units of the item stack up in the slot.
- Description Message is the message that appears when the item is examined.

Finally, there are a number of functions available for MAD items and spells. These are outlined in Table 8.17.

Table 8.17. MAD Item and Spell Functions

Function	**Purpose**
`f_use_item`	Global function to call when the item is used
`f_combine_item`	Name of function to call when the item gets used
`RemoveItemFromInventory()`	Removes items
`SetCurInvItem()`	Specifies the currently selected item
`GetCurInvItem`	Retrieves currently selected item
`GetCurInvItemID()`	Retrieves the currently selected item IDs
`AddSpellToBook()`	Adds spell to spellbook
`RemoveFrom SpellBook()`	Removes a spell from the spellbook
`GetCurSpell()`	Returns currently used spell
`GetCurSpellID`	Returns current spell ID

This makes MAD very customizable; spells and inventory items can launch any of the code already mentioned, as well as operate familiar Lua constructs.

Graphics

Lua is no slouch when it comes to graphic application integration. Lua owns a handful of open engines and even one that has been used in several commercial games. Although it is uncommon to find a completely Lua-based graphic engine, it's extremely common to find engines that rely on Lua to perform the underlying scripting.

Apocalyx 3D Engine

Apocalyx is an OpenGL 3D engine that includes Lua scripting support. The engine comes with a built-in console that can be launched and will fire Lua scripts or execute lines of Lua. The following commands are viable on the command-line console:

`h`. Reads the complete list of commands.

`l`. Reads a list of the scripts.

`r`. Executes a script.

`c`. Compiles a script.

`i`. For entering Lua lines.

Apocalyx has an entire API with exposed features and classes for Lua to manipulate. These classes are highlighted in Table 8.18, but for complete reference, check out the online manuals at the Sourceforge project page, at http://apocalyx.sf.net.

Table 8.18. The Apocalyx API

Class	Purpose	Child classes
`Background`	Used to render the sky and out of reach background objects	`HalfSky`
`Image`	Converts images and checks for alpha	`Texture`
`Material`	Holds the light properties of a surface	`BumpedMaterial`
`Reference`	Changes position and orientation of objects in 3D space	`Transform` (parent class), `Camera`, `Object`
`Sample`	Creates sounds	`Sample3D`, `Sound`, `Music`
`Simulator`	Holds physics data	`ParticleSet`
`Socket`	Holds methods for networking	`Host`
`Terrain`	Renders the ground	`Scenery`
`Win`	Manages application window	`Scene`, `World`, `Filesystem`
`Zip`	Holds methods that retrieve data from zip files	

Doris

Doris is an OpenGL viewer driven by Lua. It uses Lua, bound to OpenGL, GLUI, and GLUT, for creating graphics scripts. Doris was mainly built to perform graphical experiments, but it is also great sample code for learning how to code with Lua, OpenGL, and 3D.

Doris can be found on the Doris Sourceforge page, at http://doris.sourceforge.net. Doris was created by Nick Trout and named after his pet hamster. Currently there are versions of Doris for both Lua 4 and Lua 5. The Sourceforge page includes Lua code samples.

Nebula

Nebula is an open-source, 3D, real-time game engine written in C++. Nebula is actually scriptable through a number of languages, including both Python and Lua. It supports Direct X (8.0 and 8.1) and OpenGL and currently runs on Linux and Windows worlds.

Nebula is a base technology engine released by Radon Labs in Berlin, at http://www.radonlabs.de. Radon is responsible for a few large game products, including Project Nomads, released by CDV in 2002, and Urban Assault, released by Microsoft in 1998.

Radon is currently at work on the second generation of Nebula, Nebula2, available as a Sourceforge project. This new version of Nebula will include a new graphic system and subsystems and improvements to the code used for programming on the X-Box and will also incorporate changes made to the Nebula engine from the recent publishing of a few Radon games. Radon also plans to port Nebula to OpenGL and Linux and do a rewrite of Lua and Python support for the new engine. For more information on Nebula, check out the Nebula Wiki, at http://nebuladevice.sourceforge.net/cgi-bin/twiki/view/Nebula/.

The Games Themselves

Lua has been a part of the game industry for many years, and it probably comes as no surprise that it's been used in dozens of commercial titles. Lua can take pride in being part of many very successful products, including several that are on shelves today. In addition, a number of titles slated for release in the next few years are also jumping on the Lua bandwagon.

Angband

Angband is a freeware dungeon-exploration game based on the works of J.R.R. Tolkein (Angaband was a citadel constructed by Morgoth in Tolkein's The Silmarillion). Angband has been around in one variation or another for quite sometime. Its predecessors include Moria (1985) and Rogue (late 1970s). It was originally text-based, but now sports some nifty graphics

There are three main points to keep in mind with Angband. First, it runs on just about every platform, including Windows, Windows CE, DOS, Mac, Amiga, OS/2, Linux, BeOS, Atari, Solaris, and several others. Second, it is considered to be extremely addictive. Third, the game still fits on a 1.44 floppy disk!

Lua has been added to the C Angband distribution for customizations. There are literally dozens of Angband variants, with everything from psionics to multiplayer Iron Man adventures added. Lua scripting is available to handle using objects (like wands, rods, staves, food, potions, and scrolls) and player spells. Event handling exists for Lua functions for in-game events; for instance, Lua scripts handle which objects stores in the game will buy and sell.

Angband can be found online, at http://www.thangorodrim.net, and is currently maintained by Robert Ruhlmann.

Baldur's Gate

Bioware used Lua as the primary script engine for its popular game Baldur's Gate. All of the game's debugging commands were exposed to Lua, and the script engine was exposed and available via command line from the game. For Bioware, this allowed a deep level of debugging without having to develop extensive debugging tools for the engine. For the fans, this allowed a window into the engine that also help spawn numerous hacks and independent projects utilizing the Infinity engine.

Baldur's Gate can be found on Bioware's site at http://www.bioware.com/games/baldurs_gate/.

Bioware also used Lua to some extent in another popular game you may have heard of, MDK2.

Monkey Island

Lucas Arts was one of the first game studios to really start utilizing Lua. A large amount of Grim Fandango, the main adventure game Lucas Arts released in 1997, was written in Lua.

Lua replaced an in-house scripting engine Lucas Arts used, called SCUMM. Lua was also used in the game Monkey Island as the development script engine. In Monkey Island there is a small tribute to Lua—apparently the designers renamed a bar inside the game from SCUMM to the Lua Bar.

Homeworlds

Relic Entertainment's Homeworlds was released with Lua hooks to allow its hardcore fans the ability to create mods. The result was numerous enjoyable mods and hacks from the community, including Homeworld variants set in the worlds of Star Trek, Babylon 5, Battlestar Galactica, and Star Wars. Relic says they chose Lua for the same reasons so many other companies do: because it is easy to use, performs speedily, and is small in size.

Relic is also working on a new game that uses Lua scripts for its AI decision engine. The plan is for an interpretive AI layer to help programmers test out the different behavior easily, and therefore tweak game settings with scripting instead of having to do complete re-compiled source code builds. Relic can be found at http://www.relic.com/.

Other Games

There are dozens of other titles that have used Lua. Criterion Studios is one of the larger companies, located online at http://www.criterionstudios.com.

Criterion has released several 3D game titles here and in Japan that use Lua as their primary game scripting language. The popular fantasy RPG Pern made extensive use of Lua, so much so that the community spawned several hacks to the engine overriding some of the common Lua files that handled races and classes. Slingshot Game Technologies produced a snowboarding game using Lua called Soulride, which can be found online at http://soulride.com.

The former chief programmer at Slingshot, Thatcher Ulrich, has written a few open-source Lua 2D script tools (you used one in the last chapter). Now he works for Oddworld, which we expect to release an X-Box title any day now.

Beyond Lua

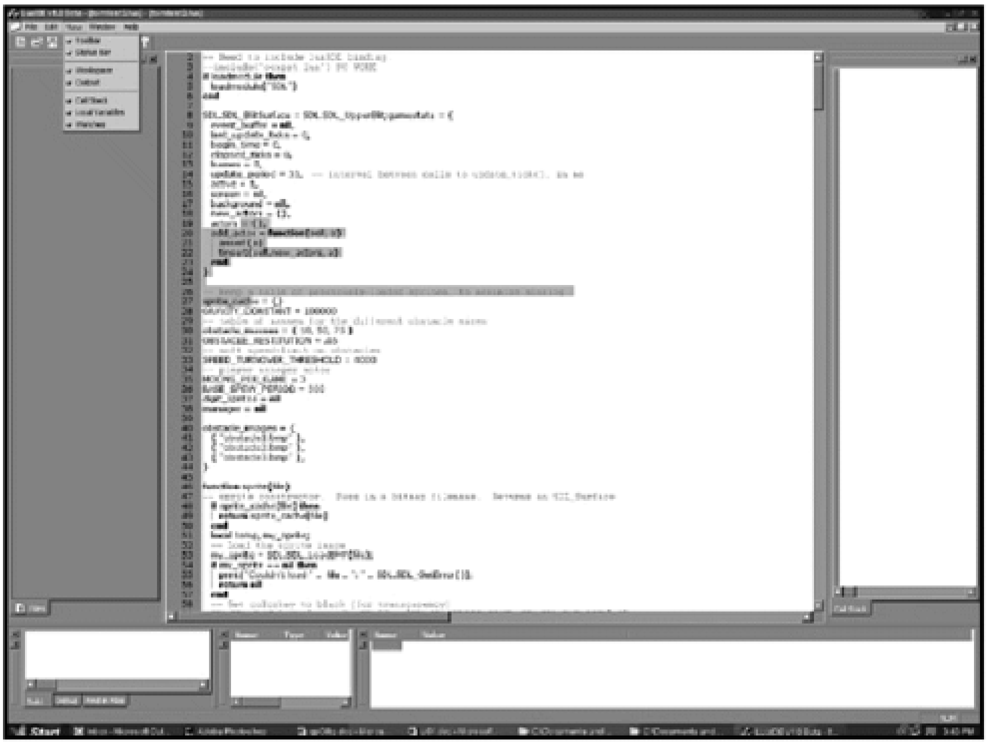
Being the versatile, lightweight creature that it is, Lua can be found in a number of different places, doing any number of different things. Not relegated to just the game world, Lua has found its way as a development language into commercial endeavors, university projects, and government agencies across the world.

LuaIDE

A programming language isn't complete until it possesses an IDE (Integrated Development Environment), and that's exactly what LuaIDE is. Developed for the community by Tristan Rybak, LuaIDE is currently (as of this writing) under a beta 1.0 release, with support for Lua 5.0. It has features for multiple-documents interfacing, Windows-build and debugging messages, breakpoints, and call stack trace windows. It also has an API for dynamically loaded Lua extensions.

Most folks familiar with a graphical development environment will recognize the interface right away (see Figure 8.3). LuaIDE can be downloaded from http://www.gorlice.net.pl/~rybak/luaide.

Figure 8.3. The Lua IDE hard at work debugging Gravity



Plua

Plua is a port of Lua to the Palm platform. Based on the PalmOS 3.1 and Lua 4.0, Plua has much to add to the platform, and is generally a complete distribution, except for a few missing pieces of functionality—mainly a few standard I/O functions, standard in

(`stdin`) functions, and math functions that would need additional support from third-party math libraries.

Despite the size restrictions for the Palm, Plua adds quite a bit to Lua distribution, including

- Database functions
- Serial input functions
- Low-level Palm graphics support
- New user interface functions for the Palm

The Plua project itself was was created and copyrighted by Mardcio M. Andrade, and documentation and sample code can be found online at http://netpage.em.com.br/mmand/pluadoc.htm.

toLua

The toLua tool is designed to make integration between Lua and C or C++ code super easy. toLua is capable of mapping C-style constants, functions, classes, variables, and methods. It also automatically generates the binding code to access these features from Lua. Version 5.0 alpha, which corresponds to the 5.0 Release of Lua, is the current release as of this writing. The software package is brought to us by Waldemar Celes, and can be found on the Lua Wiki page, at http://www.tecgraf.puc-rio.br/~celes/tolua/.

Summary

Lua is found in everything from simple 2D puzzles to complicated 3D shoot-em-up games, and from small Palm devices to large, industrial science projects. Most commonly, Lua partners with C as the script of choice to provide an additional interface of flexibility to the development team, and to add features like level builders and user customizations.

Important points from this chapter include the following:

- Lua tends to be a choice in commercial development because it is small in size, fast, and easy to use.
- SDL, C, and Lua are often partners in crime.
- Lua is fairly pervasive across the gaming industry.

Exercises

1: Use the Enigma library and sample levels to construct an Enigma level.

2: Use the sample that ships with the MAD engine to construct a MAD scene.

Part FOUR: Programming with Ruby

This part of the book begins with an overview of Ruby to get you up to speed, then moves into Ruby libraries like Rubysdl and FXRuby. Code for a quick-and-easy graphics engine written in Ruby appears in Chapter 10. Lastly, some of the more game-oriented real-life Ruby projects are discussed.

Chapter 9. Getting Started with Ruby

They brought me rubies from the mine, And held them to the sun; I said, they are drops of frozen wine From Eden's vats that run.

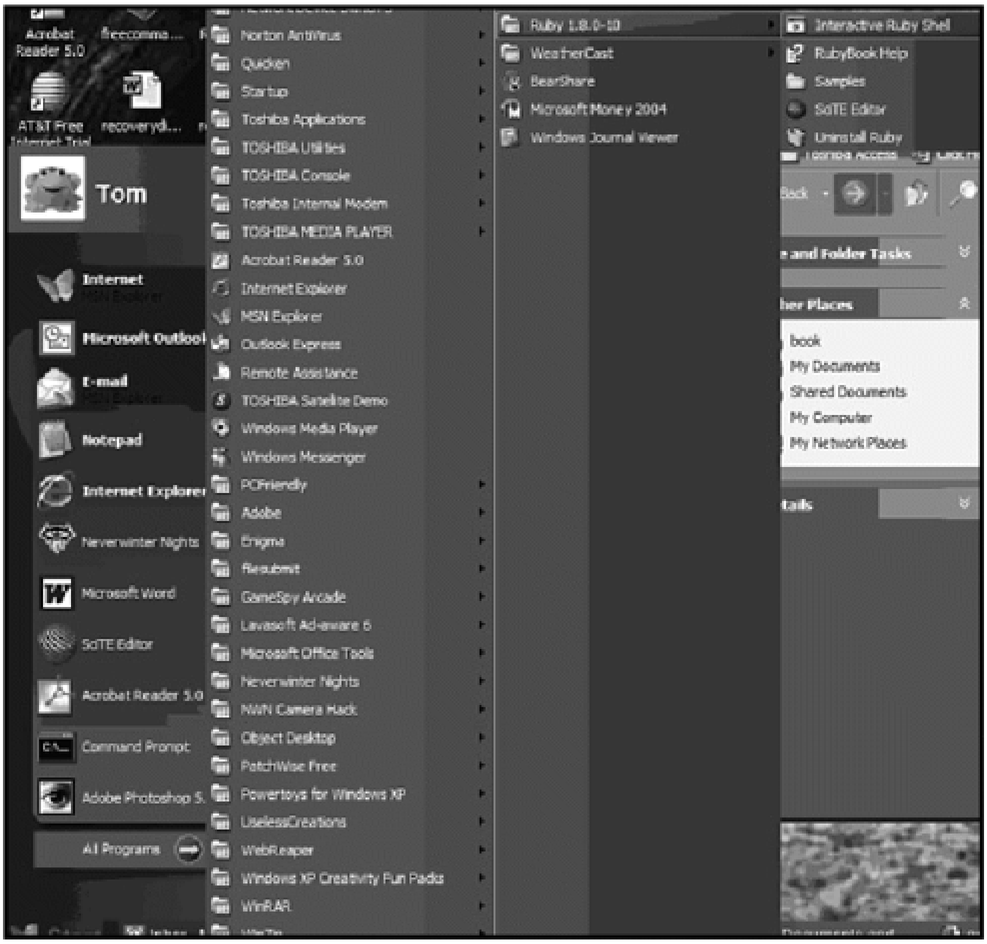
——Ralph Waldo Emerson

Like Chapters 3 and 6, this is a brief introduction to the language of interest—in this chapter, that's Ruby. This is a speedy overview chapter, but it does include a few useful examples of Ruby code.

Debuggers

Ruby comes with a debugger, accessible on the command line, for stepping through problems with programs (see Figure 9.1). Type the following to access it:

Figure 9.1. Accessing the Ruby command-line debugger




```
Ruby -r debug MyProgramScript.rb
```

The debugger has a number of useful commands for, well, debugging a Ruby program. These are listed in Table 9.1.

Table 9.1. Debug Commands

Command	Use
`break`	Set breakpoint at specified line or method
`watch`	Set a watchpoint for an expression

Table 9.1. Debug Commands

Command	Use
`catch`	Set a catchpoint for an exception
`delete`	Delete breakpoints or watchpoints
`display`	Set display expression to be printed when program stops
`undisplay`	Unset display
`cont`	Continue program execution
`step`	Step forward in the program until the next source line
`next`	Step forward in the program until the next source line. Treat method calls as one instruction
`list`	List line of source code
`up`	Select stack frame that called current stack frame
`down`	Select stack frame called by current stack frame
`finish`	Execute until selected stack frame returns
`trace`	Turn trace mode on or off
`quit`	Exit debugger
`var global`	Show global variables in current stack frame
`var local`	Show local variables in current stack frame
`var instance`	Show instance variables of the given object
`var const`	Show constants of the given object
`method instance`	Show methods of the given object
`method`	Show instance methods of the given class or module
`thread list`	Show thread list
`thread current`	Show current thread
`thread`	Switch to a given thread
`thread stop`	Stop the given thread
`thread resume`	Resume the given thread
`p`	Evaluate the given expression in current stack frame and show its value
`help`	Print debug commands
`else`	Evaluate input in the current stack frame and show its value

Language Structure

The most important thing to remember when starting out is that Ruby is considered to be a pure object-oriented scripting language. Being object-oriented means that any data itself is treated like an object. For instance, integers in Ruby automatically become objects, instances of the `number` class.

The Ruby language is considered similar to Perl and PHP. Ruby resembles Perl in a lot of ways. For instance, there are shortcuts to globals using funny characters, like `$&`, `$<`, `$>`, and `$DEBUG`. If you're familiar with string handling and pattern matching in Perl, you will find Ruby's handlings of those same problems to be similar.

One important difference between Ruby and other object-oriented languages: Ruby only supports single inheritance; most OOP languages have multiple inheritance. This means that in Ruby, sub-classes can only be derived from one parent.

A few lingual notes right off the bat. Each expression in Ruby generally takes up one line. There is no need for line terminations in Ruby. Semicolons can be used at the end of a line statement for style, but they aren't necessary. Ruby will recognize when a new line comes along, so you can end a statement by simply hitting Return. Expressions can also be grouped with parentheses.

Comments in Ruby begin with the pound sign.

```
# This is a comment.
# The interpreter ignores me
```

Comments can also be embedded between `=begin` and `=end` commands; the interpreter will also skip anything in between them:

```
=begin
This is a comment
The interpreter ignores me
=end
```

Without the equal signs, `begin` and `end` take the form of a block expression, most likely to be seen in an exception:

```
begin
# This is a block
# There are normally expressions within
end
```

Ruby supports these concepts, called blocks, which are designated in a number of different ways. Basically everything within a `do` and an `end` is a block. Blocks can also be designated with curly brackets, like so:

```
do | this_is_a_block |
end
```

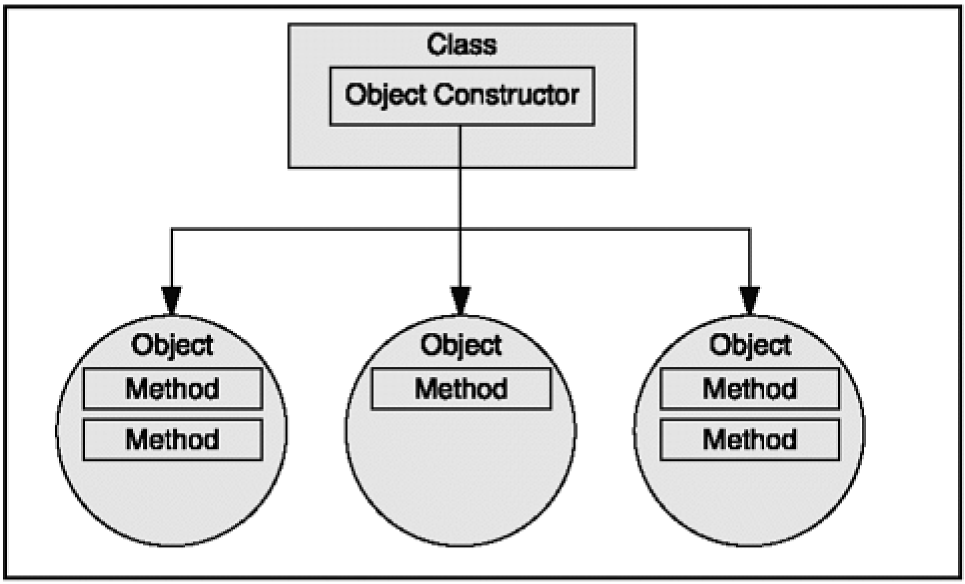
```
{this_is_another_block}
```

A special Ruby command called `yield` can call code blocks and evaluate them. `yield` evaluates the block given to the current method with arguments (if no argument is given, it uses `nil`). The argument assignment to the `block` parameter is done just like a multiple assignment.

Objects, Classes, and Methods

As Ruby is an object-oriented language, it may be useful to define some object-oriented terms. An object is a container that holds variables and functions that are specific to itself. Objects are created by classes, and are synonymous with class instances. Classes are like object factories. They combine an object template with its methods. Methods are chunks of code that return a value of some sort (see Figure 9.2).

Figure 9.2. Objects and methods derived from a class



In practice, objects, classes, and methods are combined together. One example of this is that objects are created by calling classes with their constructor methods.

In Ruby, classes take on the following form:

```
Class MyClass()
        def initialize()
        end
        def MyMethod1
        end
        # Other Expressions
        # Mayhap a CONSTANT
end
```

The `def` expressions inside of `MyClass` are actually methods:

```
Def MyMethod (arguments)
        expression
end
```

A class instance of `MyClass` (that is, a `MyClass` object) can be created by calling the `MyClass` constructor, `initialize`. An `initialize` method has special meaning in Ruby; it will automatically link up with a `new` method call. So, to create a class instance of `MyClass`, just type

```
MyObject = MyClass.new()
```

Once `MyObject` has been created, all of `MyClass`'s methods are available to it:

```
MyObject.MyMethod1
```

Ruby has a number of standard built-in classes and methods. Many of the common classes are listed in Table 9.2, and common methods are listed in Table 9.3.

Table 9.2. Built-in Ruby Classes

Classes	Domain
`Array`	Ordered collection of objects
`Bignum`	Holds integers outside the range of `Fixnum`
`Binding`	Encapsulate execution context
`Class`	Base class for all classes
`Continuation`	Objects generated by kernel call that hold a return address and execution context
`Dir`	Represents directories
`Exception`	Carries exception information
`FalseClass`	Base class for logically false
`File`	Abstraction of any file object
`File::Stat`	Common status information for file objects
`Fixnum`	Holds integer values
`Float`	Represents real numbers
`Hash`	Collection of key value pairs
`Integer`	Base class for `Bignum` and `Fixnum`
`IO`	Basis for all input and output
`MatchData`	Type of the special `$~` variable for encapsulating pattern matches

Table 9.2. Built-in Ruby Classes

Classes	Domain
`Method`	Base class for method objects
`Module`	Base class for module objects
`NilClass`	Base class for `nil`
`Numeric`	Base type for `Float`, `Fixnum`, and `Bignum`
`Object`	Parent class for all classes
`Proc`	Parent for object blocks of code that are bound to a set of local variables
`Range`	An interval with a start and end
`Regexp`	Holds regular expressions
`String`	Holds byte sequences
`Struct`	Bundles attributes together
`Struct::Tms`	Holds information on process times
`Symbol`	Object that represents a Ruby name
`Thread`	Parent for thread objects
`ThreadGroup`	For keeping track of threads as a group
`Time`	Abstraction of dates and times
`TrueClass`	Base class for logically true

Table 9.3. Built-in Ruby Methods

Method	Description/What It Does
`'str`	Performs `str` by a subshell
`Array`	Converts given argument to an array
`at_exit`	For cleaning up when the interpreter exits
`autoload`	Specifies a file to be loaded using `require`
`binding`	Returns data structure of a binding
`caller`	Returns the information of the current call
`catch`	Executes a `throw/catch` block
`chop`	Removes last character of a value
`chomp`	Removes a line from a value
`eval`	Evaluates the given expression as a Ruby program
`exec`	Executes the given command as a subprocess
`exit`	Exits immediately

Table 9.3. Built-in Ruby Methods

Method	Description/What It Does
`exit!`	Exits immediately and ignores any kind of exception handling
`fail`	Raises exceptions
`Float`	Converts given argument to a float
`fork`	Forks child and parent processes
`format`	Returns a string in the given format
`gets`	Reads a string from standard input
`global_variables`	Returns the list of all the global variable names defined in the given program
`gsub`	Searches for a pattern in a string, and if the pattern is found, makes a copy of the string with the pattern replaced with a given argument
`gsub!`	Searches for a pattern in a string, and if the pattern is found, replaces the pattern with the given argument.
`Integer`	Converts given argument to an integer
`iterator?`	Returns true if called from an iterator
`lambda`	Returns newly created procedure object from the block
`load`	Loads and evaluates the Ruby program in the file
`local_variables`	Returns the list of all the local variable names defined in the current scope
`loop`	Loops until terminated
`open`	Opens a file and returns a file object associated with the file
`p`	Prints given object to the `stdout`
`print`	Prints arguments
`printf`	Prints arguments in a given format
`proc`	Returns newly created procedure object from the block
`putc`	Writes a given character to the default output
`raise`	Used for raising exceptions
`rand`	Returns a random number greater than or equal to 0 and less than the given value
`readline`	Reads a string from standard input, raises exception at the end of a file
`readlines`	Returns an array containing the read lines
`require`	Used to load modules
`select`	Calls `select` to reads, writes, excepts, and timeout system calls

Table 9.3. Built-in Ruby Methods

Method	Description/What It Does
`sleep`	Causes script to sleep for a given amount of seconds
`split`	Splits a string at the given string
`String`	Converts given argument to a string
`sprintf`	Returns a string in the given format
`srand`	Sets the random number seed for `rand`
`sub`	Searches a string held in `$_` for a pattern and makes a copy that replaces the first occurrence with given argument
`sub!`	Searches a string held in `$_` for a pattern and replaces the first occurrence with given argument
`syscall`	Used to make system calls
`system`	Runs given command in a subprocess
`test`	Performs a file test
`throw`	Executes a `throw/catch` block
`trace_var`	Sets the hook to a given variable that is called when the value of the variable is changed
`trap`	Specifies the signal handler for a signal
`untrace_var`	Deletes hooks set by `trace_var`

Obviously, Ruby supports inheritance. Inheritance allows classes to inherit functionality from each other. A parent class in this sort of relationship is called a super class, and the child class is called a sub class. Sub classes are defined as children by the less than (<) symbol:

```
MySubClass < MyParentClass
        def MyMethod (arguments)
        expression
end
```

Ruby also supports a number of other OOP concepts, such as mixins, which simulate multiple inheritance in Ruby's single parent system); singletons, which provide a way to override object creation; and overloading, which is when method calls can be overwritten by new definitions.

Language Types

Ruby is case-sensitive, and identifiers can be composed of letters of the alphabet, decimals, or the underscore character. The standard language types include strings, constants, ranges, and numbers.

All variables and constants in Ruby point at an object. When a variable is assigned or initialized, the object that the variable is referencing is also assigned. Variables in Ruby are either global variables that begin with the `$` character, instance variables that begin with an `@` character, class variables that start with `@@`, constants that are all uppercase letters, local variables that are all lowercase letters, or class constants that are defined within certain classes or modules. There are also a few special variables in Ruby. All of these Ruby variables are outlined in Table 9.4.

Table 9.4. Ruby Variables

Variable	Description
`__FILE__`	Current source filename
`__LINE__`	Current line number in the source file
`@variable`	Instance variable
`@@`	Class variable
`$variable`	Global variable
`variable`	Standard variable
`VARIABLE`	Constant variable
`false`	Instance of the class `FalseClass` (i.e. false)
`nil`	Instance of the class `NilClass` (i.e. false)
`self`	Receiver of the current method
`true`	Instance of the class `TrueClass` (that is, true)

Setting a variable is fairly intuitive; it is done like so:

```
myvariable = "My String\n"
```

Set a global variable like this:

```
$myvarible = "My String\n"
```

Set class variables like this:

```
@@myvariable = "My String\n"
```

And so on.

In Ruby, a handful of reserved words cannot be used for variable names. These include the following:

```
BEGIN
```

```
class
ensure
nil
self
when
END
def
false
not
super
while
alias
defined
for
or
then
yield
and
do
if
redo
true
begin
else
in
rescue
undef
break
```

```
elsif

module

retry

unless

case

end

next

return

until
```

Strings

In Ruby, strings are 8-bit byte sequences. They normally hold characters, but can also hold binary data. A string object is actually an instance of the class `String`.

When playing with strings, one should know that Ruby differentiates between single and double quotes. Notice the difference between

```
print "string\n"
```

and

```
Print 'string\n'
```

When the two lines of code above run within the interpreter, Ruby recognizes the `\n` as an escape sequence on the first line and not on the second.

The explanation for this is that strings can begin and end with either single or double quotation marks, but whether you use single or double quotation marks depends on the situation. Expressions in double quotes are generally subject to backslash escape characters, and single quotes are not (except for `\` and `\\`). If you need a string to not be subject to any escape sequences, including the `\` or `\\`, then you must begin the expression with a percentage `%` sign.

The `String` class has many methods (close to 100) associated with it. Since Ruby must often handle strings, it makes sense that it would have many standard methods for performing standard actions on strings.

Common escape sequences are listed in Table 9.5.

Table 9.5. Common Ruby Escape Sequences

Sequence	Meaning
`\`	Add octal value character
`\a`	Bell
`\b`	Backspace
`\cx`	Control x
`\C-x`	Control x
`\e`	Escape
`\f`	Form feed
`\n`	Newline
\r	Carriage return
`\s`	White space
`\t`	Tab
`\v`	Vertical tab
`\x`	Add hexadecimal value character
`\M-x`	Meta x
`\M-\C-x`	Meta control x

Regular Expressions

Regular expressions are the tools that Ruby uses for pattern matching and other common functions against strings. Not only do commands exist for matching patterns, but there are also commands for anchoring patterns, repeating searches, choosing between alternate patterns, grouping, and substitution. Common Ruby expressions are listed in Table 9.6.

Table 9.6. Common Ruby Expressions

Expression	Description
`$!`	Exception information message (set by `raise`)
`$@`	Backtrace of the last exception
`$&`	The string matched by the last successful pattern match in this scope
`` $` ``	The string preceding whatever was matched by the last successful pattern match in the current scope
`$'`	The string following whatever was matched by the last successful pattern match in the current scope
`$+`	The last bracket matched by the last successful search pattern

Table 9.6. Common Ruby Expressions

Expression	Description
`$1, $2...`	Contains the subpattern from the corresponding set of parentheses in the last successful pattern matched
`$~`	Information about the last match in the current scope
`$=`	Flag for case insensitive
`$/`	Input record separator
`$\`	Output record separator
`$,`	Output field separator
`$;`	Default separator for `String#split`
`$.`	The current input line number of the last file that was read
`$<`	The virtual concatenation file of the files given by command-line arguments. `Stdin` by default
`$>`	Default output for print, `printf.s` by default
`$_`	The last input line of string by `gets` or `readline`
`$0`	Contains the name of the file containing the Ruby script being executed
`$*`	Command-line arguments given for the script
`$$`	The process number of Ruby running this script
`$?`	The status of the last executed child process
`$:`	The array contains the list of places to look for Ruby scripts and binary modules by `load` or `require`
`$"`	The array contains the module names loaded by `require`
`$DEBUG`	Status of the `-d` switch
`$FILENAME`	Same as `$<.filename` or `stdin` filename
`$LOAD_PATH`	The alias to the `$:`
`$stdin`	The current standard input
`$stdout`	The current standard output
`$stderr`	The current standard error output
`$VERBOSE`	The verbose flag, which is set by the `-v` switch to the Ruby interpreter
`$-0`	The alias to the `$/`
`$-a`	True if option `-a` is set
`$-d`	The alias to the `$DEBUG`
`$-F`	The alias to the `$;. \`
`$-i`	In in-place-edit mode
`$-I`	The alias to the `$:`

Table 9.6. Common Ruby Expressions

Expression	Description
`$-l`	True if option `-lis` set
`$-p`	True if option `-pis` set
`$-v`	The alias to the `$VERBOSE`

Constants

Like variables, constants hold references to objects and are created when they are first assigned. Unlike in other languages, constants can be changed in Ruby, although a change fires off a warning from the interpreter.

Constants that are defined within a class or module are accessible from within the class or module. Outside of the class or module, the scope operator `(::)` can be used to access them. Ruby also has a number of pre-defined constants that are global in scope; these are listed in Table 9.7.

Table 9.7. Ruby Global Pre-Defined Constants

Constant	Description
`ARGF`	Alias to `$<`
`ARGV`	Alias to `$*`
`DATA`	The file object of the script
`ENV`	Contains current environment variables
`FALSE`	False
`NIL`	Nil
`RUBY_RELEASE_DATE`	Ruby release date string
`RUBY_PLATFORM`	Ruby platform identifier
`STDIN`	Standard input or `$stdin`
`STDOUT`	Standard output or `$stdout`
`STDERR`	Standard error or `$stderr`
`TRUE`	True
`VERSION`	Ruby version string

Ranges

Ranges are used in Ruby to express sequences such as one through ten or A to Z. Ranges come with a number of useful methods for iterating over themselves or testing their contents. The `..` operator is used to create a range type object:

```
myrange = 1..10
```

Ranges are probably most often used to create arrays but are also sometimes used in conditional statements.

Numbers

Ruby deals primarily with integers and floating-point numbers. Smaller numbers are objects of the `Fixnum` class and large numbers are objects of the `Bignum` class. Ruby understands octal, binary, and hexadecimal numbers. Ruby numbers are outlined in Table 9.8.

Table 9.8. Ruby Numbers

Number	Type
`10`	Integer
`-10`	Signed integer
`10.10`	Floating point number
`0xffff`	Hexadecimal integer
`0b01011`	Binary integer
`0377`	Octal integer

As all numbers are objects, you may find them used in Ruby in what would be unusual ways in other languages. Numbers in Ruby will respond to messages and built-in methods that do iterations.

Ruby has a number of operators, including the standard +, -, /, and *. Again, most operators are method calls. Following is a list of the Ruby operators, from highest to lowest level of precedence (the operators are further outlined in Table 9.9):

1. ::
2. **
3. -(unary) +(unary) ! ~
4. / %
5. + -
6. << >>
7. &
8. | ^
9. >= < <=
10. <=> == === != =~ !~
11. &&

12. ||
13. .. ...
14. =(+=, -=...)
15. not
16. and or

Table 9.9. Commonly Used Ruby Operators

Operator	Use
`==`	Tests for equality
`===`	Tests equality in a `case` statement
`<=>`	Compares two values
`<, <=, >=, >`	Less than, greater than, etc.
`=~`	Regular expression pattern match
`+= 1`	Increment by 1
`-=1`	Decrement by 1
`&&`	Logical and
`||`	Logical or
`!`	Logical not
`!=`	Not equal
`..`	Range
`::`	Scope resolution
`and`	Logical and
`eql?`	Compares type and value
`equal?`	Compares object ID
`not`	Logical not
`or`	Logical or

Assignments are made in Ruby using the powerful equal sign:

```
Hello = 'Hi'
```

Multiple assignments can be made by using commas and equal signs:

```
1,2,3 = 'you', 'me' , 'I'
```

Control Structures

Ruby has a number of standard control structure expressions for controlling program flow. These include `if`, `then`, `unless`, `else`, `while`, `until`, `for`, and `case`. These controls are supported by a number of operators in addition to the standard ==, <, <=, >, >=, and =. These operators are listed in Table 9.9.

The chain of `if… then… else…` is one of the most common control structures, and in Ruby the syntax appears as follows:

```
if this_is_true [then]
         do_this
        [elsif this_is_true_instead [then]
          do_this_Instead]
        [else
          do_this_if_all_else_fails]
        end
```

A typical `unless` control structure looks almost identical:

```
unless this_is_true [then]
          do_this
        [else
          do_this_instead]
        end
```

The `case` statement in Ruby is much like a quicker-coding `if` statement when multiple choices are available. The `case` statement makes a comparison between the expression given and any number of expressions (or ranges) that are set after `while` keywords:

```
case $my_case_statement
when 0 .. 1 "case_1"
when 2 .. 3 "case_2"
when 4.. 20 "case_3"
when Square "case_4_sides"
else "case_5"
end
```

The `until` construct and the `while` construct are also very similar:

```
# First until
until this_is_true
  do_this
end
#Then while
while this_is_true
   do_this
end
```

The `for` looping construct is used to iterate over any object that can respond to iteration, namely arrays or ranges. The following example prints everything listed in the first argument given:

```
for i in [1, 2, 3]
  print I, " "
end
```

There are a few useful commands that can be used in loops (and in blocks), including the following:

break. Terminates the immediately enclosing loop.

next. Skips to the end of the loop.

redo. Repeats the current loop from the start without re-evaluating the condition.

retry. Repeats the current loop.

return. Exits the method/loop/block with the return value or an array of return values.

For handling exceptions Ruby has a built-in `raise` command. `raise` can create a runtime error, send messages, create an exception of type `error_type`, and even send traceback information in the format given by a variable caller function. Examples:

```
raise "This is an error"
raise SyntaxError, "invalid syntax"
raise SyntaxError.new("new error type")
```

Arrays and Hashes

Arrays are instances of the class `Array`, and they hold collections of object references. An array can be created by simply assigning a number of values as you would with a variable:

```
$MyArray = ['Hello', 'I', 'love', 'you']
```

Each value in the array can then be accessed numerically:

```
print $MyArray[0]
print $MyArray[1]
print $MyArray[2]
print $MyArray[2]
```

Hashes are also instances of the `Hash` class, and are also collections of object references. Hashes are like arrays, only within curly brackets and with key => value pairs:

```
$MyHash = {'a'=>1, 'b'=>2, 'c'=>3, 'd'=>4}
```

Then hash values can be referenced by their keys:

```
print $MyHash['a']
print $MyHash['b']
print $MyHash['c']
print $MyHash['d']
```

Exceptions

Ruby has built-in `exception` classes for raising exceptions, each of which has its own message string and stack backtrace. Exception code is normally put into `begin` and `end` blocks, and is handled by a `rescue` clause, like this:

```
begin
         # code that does something
rescue Exception
        $stderr.print "error message"
        raise
end
```

The `raise` method is used in this case to raise the current exception, but it can also be used to create a unique exception and error message:

```
if MyError == true
raise MyError, "My Unique Error", caller
end
```

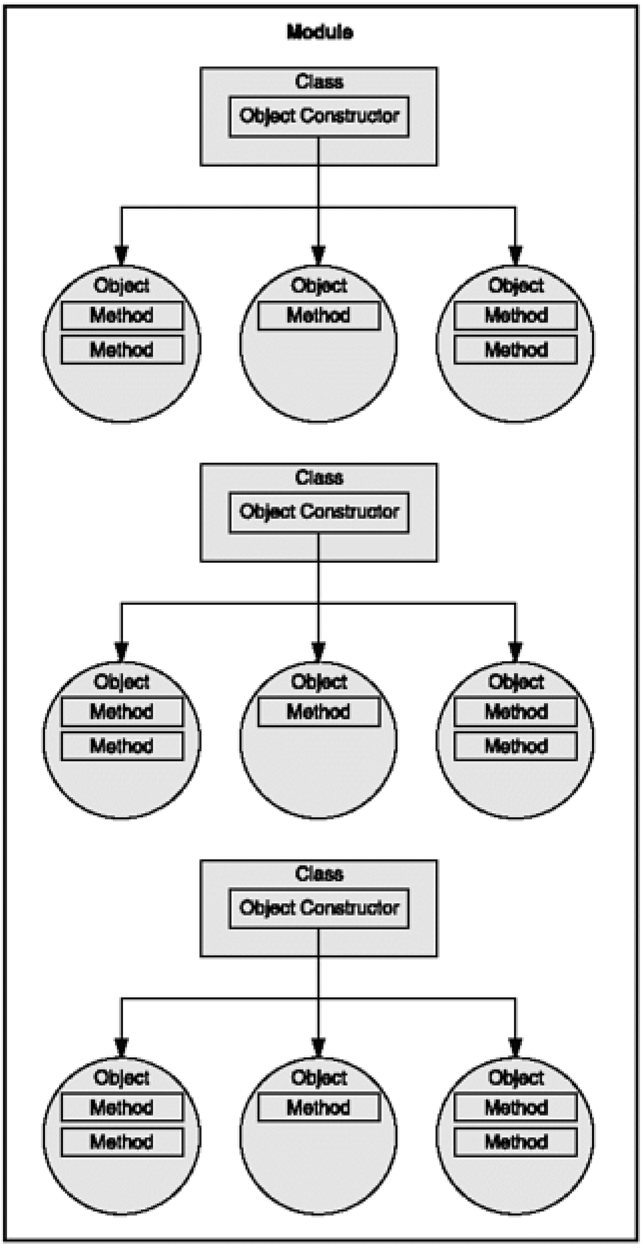
`Catch` and `throw` are also used when execution must be abandoned completely. The `catch` command creates a block of code that executes normally until a `throw`is encountered. When Ruby hits a `throw`, it goes back up the call stack looking for the matching `catch`, rewinds to that point, and terminates the block. Optionally, when Ruby jumps back up the stack, another `throw` parameter can be sent upwards, causing Ruby to continue bouncing upward:

```
catch (:MyCatch) do
        while (1)
                throw :MyCatch unless all_is_well
        end
end
```

Modules

Modules are groups of methods, classes, and constants (see Figure 9.3). As programs grow bigger and bigger, it becomes necessary for most languages to segregate bits of functionality and similar functions. Modules are Ruby's way of organizing large batches of code.

Figure 9.3. Modules hold groups of classes and methods



Modules can be defined fairly easily with the `module` command:

```
module MyModule
        SOME_CONSTANT=1
        ANOTHER_CONSTANT=2
        def Some_Class
        # class expressions
end
end
```

Save the module into a file called MyFile.rb, and Ruby will have the option of loading the additional module by using `require` or `load`:

```
Load "MyFile.rb"
Require "MyModule"
```

Ruby comes with a number of modules for adding extra functionality to your program already built in (see Table 9.10).

Table 9.10. Pre-Defined Ruby Modules

Module	Use/Description
`Comparable`	Comparing objects
`Enumerable`	Traversal and searching methods
`Errno`	Mapping OS system errors
`FileTest`	File test operations
`GC`	Garbage collection interface
`Kernel`	Objects can access kernel module
`Marshal`	Ability to serialize objects
`Math`	Basic trigonometry and transcendental functions
`ObjectSpace`	Added GC functionality for iterating over all living objects
`Precision`	Number precision
`Process`	Manipulate processes

Libraries

Libraries are collections of modules and classes, and Ruby has a wealth of them. Table 9.11 lists a few of the more common libraries and their general purposes. These libraries are written in Ruby itself and are found in the /lib/ folder of the distribution.

Table 9.11. Ruby Libraries

Library	Purpose/Description
`Delegate`	For building delegate-type objects
`English`	Includes the English library file
`Observer`	For communication between objects
`Profile`	For code profiling, prints summary of system calls to `$stderr`
`Network`	Ruby provides a number of socket-level access classes, including `socket`, `BasicSocket`, `IPSocket`, `TCPSocket`, `SOCKSocket`, `TCPServer`, and `UDPSocket`

Table 9.11. Ruby Libraries

Library	Purpose/Description
`Singleton`	For ensuring that only one instance of a particular class is created
`Timeout`	For timing code blocks

Memory, Performance, and Speed

Ruby suffers from the same speed impediment as the other languages in this book—being interpretive and-high level. Ruby's built-in profiler tool helps quite a bit when gauging performance by examining code snips and routines for slowdowns. The profiler can be called from the command line by adding `-r profile`, or it can be used inside code by using `require`, like so:

```
require "profile"
```

The profile library will print a summary of the number of calls to each Ruby method in the given program and the time spent in each method. This is all printed to `$stdout`.

Garbage Collection

Ruby has a mark and sweep garbage collection system. It periodically sweeps through dynamically allocated memory and reclaims it if it isn't in use. Ruby also provides a GC (garbage collection) module for interfacing the underlying garbage collection methods, which include

- **disable.** Disables garbage collection.
- **enable.** Enables garbage collection.
- **start.** Initiates a garbage run (unless disabled).
- **garbage_collect.** Starts garbage collector.

Speed

Finally, here are a number of tricks you can use to help speed up Ruby code:

Check the profiler to see where the code is bogging down.

Use the built-in GC to take control over garbage collection.

Initialize variables before they are used. Variables used within a block can be defined before the interpreter hits the block.

When iterating over a large set of elements, declare any iterator variables.

When returning variables from a block, have the variables pre-initialized so they aren't allocated on-the-fly.

Ruby supports both multiple threads and forks for creating sub-processes. Threads are implemented within the interpreter, and forks are invoked in the operating system. Either of these can be used for a speed hit in certain cases.

Summary

Before moving on to the next chapter, you should have Ruby installed on your computer, and you should feel quite comfortable using Ruby blocks, classes, methods, variables, and common control structures like `while` and `if`.

Important points from this chapter:

- Any form of data in Ruby is an object.
- An object is a container that holds variables and functions that are specific to itself.
- Lowercase letters equal local variables, uppercase letters equal a constant, variables that start with `$` are global, variables that start with `@` are instance variables, and variables that start with `@@` are class variables.
- End-of-line deliminators are not necessary in Ruby.

Questions and Answers

1: Q: Ack! What does "parse error" mean?

A: A: It usually means you have a block that's missing an `end`.

2: Q: I heard that Ruby is a compiled language and not interpreted; is this true?

A: A: There are plans in the future to move Ruby closer to a compiled language in order to increase its execution speed.

Exercises

1: Define object, class, and method.

2: What is the difference between a Ruby hash and a Ruby array? How is each declared?

3: Describe two built-in Ruby classes.

Chapter 10. Getting Started with Ruby Games

The price of wisdom is above rubies.

——Job 28:18

This chapter covers the common libraries used for game programming in Ruby, focusing on Ruby's OpenGL and SDL wrappers.

FXRuby

Ruby comes with a few toolkits, including FXRuby and OpenGL. FXRuby is a Ruby interface to the FOX toolkit, which is designed for creating graphical user interfaces and which is written in C++.

The Fox toolkit has a home at http://www.fox-toolkit.org/.

FXRuby and OpenGL for Ruby are partners in many projects. They are often used together, and Fox provides a few widgets for providing OpenGL support. `FXGLCanvas` and `FXGLViewer` are `FXGLVisual` objects. `FXGLVisual` can be used to create new visual applications and includes options for double buffering (`VISUAL_DOUBLEBUFFER`) and stereo sound (`VISUAL_STEREO`):

```
MyVisualObject = FXGLVisual.new(MyApplication, VISUAL_STEREO)
```

The `FXGLCanvas` widget is an OpenGL window with minimal functionality:

```
MyCanvas = FXGLCanvas.new (MyApplication, visual_to_use)
```

The `FXGLViewer`widget is a higher-level OpenGL window with more functionality and is built the same way:

```
MyViewer = FXGLViewer.new (MyApplication, visual_to_use)
```

There are a number of important differences between the standard FOX API and the actual FXRuby API. FXRuby uses Ruby strings instead of the standard Fox FXStrings. Since Ruby handles underlying memory management, some of the drudgery of handling pointers and arrays in FOX can be skipped. Many of the FOX classes have been extended with built-in Ruby methods such as `each`, `initialize`, and `catch`. There are also differences in multi-threading and the return values to a few interfaces.

With the FXRuby included in the standard Ruby package comes a number of FXRuby samples. These are in the (surprise!) Samples directory. FXRuby is fairly easy to include in a Ruby script. Once installed, Ruby's `require` and `include` commands can be used to bring in the library:

```
#!/usr/bin/env ruby
require "fox"
include Fox
```

A new FXRuby application is declared by calling the `FXApp` class with its `new` constructor:

```
MyApplication = FXApp.new()
```

FXRuby requires a main or parent window; these can be declared in the same way using the new method of `FXMainWindow`. This function ties into the newly created `MyApplication` Fox application and is also fed the window title:

```
main = FXMainWindow.new(MyApplication, "FXWindow")
```

You can create a GUI button in the main window by using `FXButton`:

```
FXButton.new(main, "Press This Button!")
```

Finally, you must create `MyApplication` with a `create` method, show it on the screen with a `show` method, and then turn it on with a `run` method:

```
MyApplication.create()
main.show(PLACEMENT_SCREEN)
MyApplication.run()
```

Ruby and OpenGL

I discussed OpenGL first in Chapter 4, where I introduced PyOpenGL. To briefly summarize, OpenGL is a platform-independent API for creating graphics. Ruby's OpenGL extension module was developed by Yoshiyuki Kusano. It provides an interface to the basic OpenGL, GLU, and GLUT APIs.

As of this writing, the extension is at Version 0.32b and can be found at Yohshiyukani Kusabo's homepage, at http://www2.giganet.net/~yoshi.

GLU is a high-level library that partners with OpenGL. It provides additional functionality that would otherwise be fairly difficult to code in just OpenGL. GLUT is another OpenGL addition—it's a toolkit for designing OpenGL programs. Together these two build an API that allows Ruby to easily access OpenGL commands.

A simple example of OpenGL is drawing a geometric shape. Step 1 is including the OpenGL, GLU, and GLUT libraries if they are necessary. Posix systems may also need the Ruby path—that is, `#!/usr/local/bin/ruby`:

```
#!/usr/local/bin/ruby
require "opengl"
require "glut"
```

Using the OpenGL `Proc` function and its `new` method will declare a new function, called `MyTriangle`, that can draw a geometric shape:

```
MyTriangle = Proc.new {
```

In order to draw the shape, the GL buffer must be cleared, and a new GL object of type `TRIANGLE` must be created:

```
GL.Clear(GL::COLOR_BUFFER_BIT)
GL.Begin(GL::TRIANGLES)
```

Then you can set, with `GL.Color`, the RGB values that you'll use when drawing:

```
GL.Color(1.0, 1.0, 1.0)
```

And then you need to set the vertices of the three points of the triangle in 2D space:

```
GL.Color(1.0, 1.0, 1.0)
GL.Vertex(0, 0)
GL.Vertex(10, 10)
GL.Vertex(10, 50)
```

The OpenGL buffer must be flushed with `GL.Flush` and the calls to OpenGL ended.

The whole `MyTriangle` function looks like this:

```
MyTriangle = Proc.new {
  GL.Clear(GL::COLOR_BUFFER_BIT)
  GL.Begin(GL::TRIANGLES)
    GL.Color(1.0, 1.0, 1.0)
    GL.Vertex(0, 0)
    GL.Vertex(10, 10)
    GL.Vertex(10, 50)
  GL.End
  GL.Flush
```

In order to use the function, you must create a window (`MyWindow`) for display. The window can be built using GLUT's `CreateWindow` method after GLUT is initialized:

```
GLUT.Init
MyWindow =  GLUT.CreateWindow("OpenGL Triangle")
```

The final steps for actually running this short Ruby OpenGL sample are to use GLUT's `DisplayFunc` method to display `MyTriangle`, and then call `MainLoop` to get it all started:

```
GLUT.DisplayFunc(MyTriangle)
GLUT.MainLoop
```

The standard Ruby install comes with many OpenGL samples located in the Ruby\Samples\OpenGL directory. These include examples showing how to draw two-dimensional and three-dimensional shapes, play with colors, and rotate objects in three-dimensional space.

Ruby and SDL

SDL has been a common thread throughout the book, first in Chapter 4 with the Pygame SDL wrapper for Python, and then in Chapter 7 with LuaSDL. It would stand to reason that SDL, being the progressive library that it is, also has its fingers in Ruby.

To use SDL with Ruby, you first need to install the SDL library, which can be found in its entirety at its home page, http://www.libsdl.org.

Once SDL is installed, Ruby needs an interface into SDL; there are several different interfaces to choose from. Most of the interfaces can be found in the Ruby Application Archive at http://raa.ruby-lang.org/.

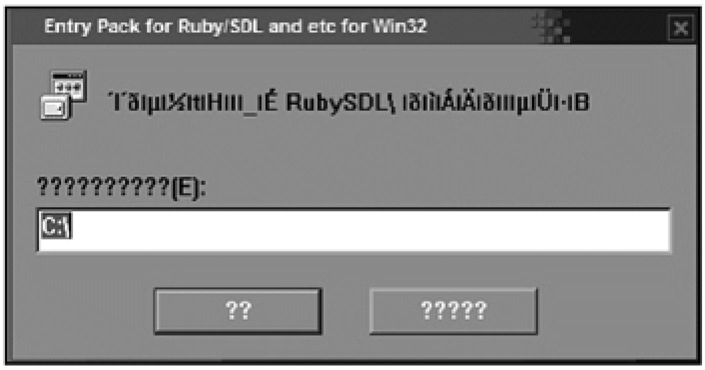
To make it a bit easier for Windows users, a bundled SDL package is contained in a nifty executable included on this book's CD; it is called pack-rubysdl.exe, and you can find it on the CD in the Chapter 10 folder. The pack-rubysdl.exe package is distributed under the GNU Public License and, for Win32, includes the following:

- **Ruby 1.6.4.** The Ruby version.
- **Rubysdl-0.6.** The actual SDL package.
- **Rubywin-0.0.3.2.** An IDE for Ruby on Windows platforms.
- **Rb2exe-0.2.** A program for converting Ruby scripts into executable files.
- **Opengl-0.32.** The version of OpenGL.

The package is built with Cygwin and comes with a few SDL samples, including those for using the keyboard and joystick, loading sound files from disk, and manipulating a CD. The package also includes one fairly complete sample game by Ohbayashi Ippei.

The caveat to this bundle is that the documentation and installation are in Japanese. You will not be able to read the install files without the proper Japanese character set installed. This is inconvenient for English speakers, as the install files may look like Figure 10.1, depending on the platform used.

Figure 10.1. The pack-rubysdl install may look strange without the right Japanese character set.



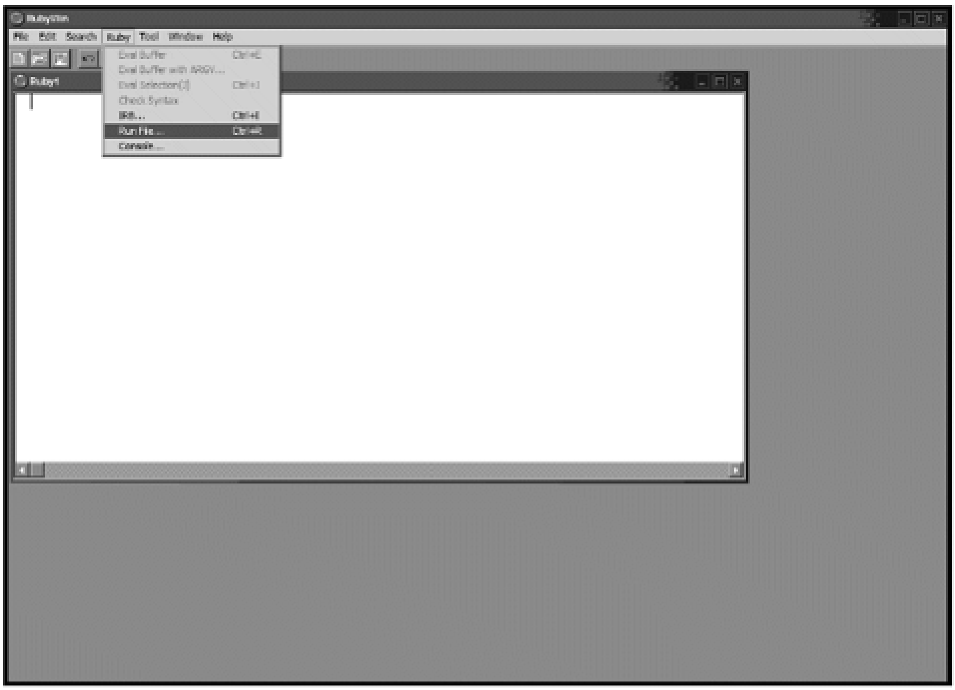
Whether your platform displays the characters correctly or not, choosing the left-hand confirmation button means you agree to install the RubySDL folder and files on your C:\ drive (see Figure 10.2).

Figure 10.2. Choosing the left-hand button at this screen after launching .pack-rubysdl.exe will install the package.



RubyWin is one of the big bonuses in this package. A GUI developed by Masaki Suketa that bundles Ruby 1.6.4 and Scintilla 1.38 (by Neil Hodgsen), RubyWin creates an environment for running Ruby SDL scripts without having to change or manipulate local environment variables. Launching the executable brings up the RubyWin GUI (see Figure 10.3) and the Run File command, accessible via the Ctrl-R shortcut or through the Ruby menu, can be used to launch and test Ruby SDL applications

Figure 10.3. The RubyWin GUI



Commonly Used Ruby SDL Modules and Classes

The common Ruby SDL modules and classes are listed in Table 10.1.

Table 10.1. Common Ruby SDL Modules and Classes

Component	Module or Class	Description
`SDL::CD`	Class	Represents the CD-ROM drive
`SDL::Error`	Class	Error class; handles Ruby/SDL errors
`SDL::Event`	Class	Handles events
`SDL::Event2`	Class	Handles events
`SDL::Joystick`	Class	Represents a joystick
`SDL::Key`	Module	Defines key constants and gets the key state
`SDL::Mixer`	Module	Holds sound functions and constants
`SDL::Mixer::Wave`	Class	Handles WAV files
`SDL::MPEG`	Class	Handles MPEG streams
`SDL::Mouse`	Module	Contains mouse constants and functions
`SDL::PixelFormat`	Class	Parent to `SDL::Surface` (obsolete)
`SDL::Screen`	Class	Displays the screen image
`SDL::SKK`	Class	Handles Japanese input
`SDL::Surface`	Class	Contains methods for creating SDL surfaces (images)
`SDL::TTF`	Class	Handles TrueType fonts
`SDL::WM`	Module	Handles windows

Ruby SDL includes all sorts of classes for supporting window management, MPEG streaming, joysticks, CD-ROMs, and different fonts. More commonly used are the tools for initializing and SDL environments, creating SDL surfaces, handling events, audio, time, and Japanese character support.

Initializing SDL

The `init` module is used to initiate SDL. A flag that triggers which portion of SDL needs to be initialized is included when initializing:

SDL::INIT_AUDIO. Initialize system audio.

SDL::INIT_VIDEO. Initialize system video.

SDL::INIT_CDROM. Initialize the CD-ROM.

SDL::INIT_JOYSTICK. Initialize a joystick device.

The line of code that will initialize video looks like the following:

```
SDL.init(SDL::INIT_VIDEO)
```

A particular game's video mode is set with `SDL.set_video_mode()`, which takes as arguments the width and height of the screen, bits-per-pixel (0=s current or local display), and any necessary flags:

```
SDL.set_video_mode(640, 480, 0, SDL_FLAG)
```

Possible flags for `SDL.set_video_mode` include the following:

SDL::SWSURFACE. Creates video surface in system memory.

SDL::HWSURFACE. Creates video surface in video memory.

SDL::FULLSCREEN. Attempts to use the full screen.

SDL::SDL_DOUBLEBUF. Enables double buffering.

To find out if a particular video mode is supported, there is also an `SDL.checkVideoMode()` command that uses the same syntax.

Surfaces

After setting up a video mode, `SDL::Surface.new` will create an empty SDL surface. Its `new` method also keeps an eye out for several flags:

SDL::SWSURFACE. Creates the surface in system memory.

SDL::HWSURFACE. Creates the surface in video memory.

SDL::SRCALPHA. Chooses the location with the best hardware alpha support.

SDL::SRCOLORKEY. SDL chooses the location with the best hardware colorkey blitting.

`Surface.new` also needs width, height, and format. The format must be the instance of `SDL::Surface` and have the same bits per pixel as the specified surface.

There are dozens of methods that can be used on SDL surfaces. Some of the more commonly used ones are listed in Table 10.2.

Table 10.2. Common SDL Surface Methods

Method	**Equivalent To**	**Purpose**
alpha		Returns surface alpha
bpp		Return bits per pixel
colorkey		Returns surface colorkey
drawCircle	draw_circle	Draws a circle
drawEllipse	draw_ellipse	Draws an ellipse
DrawFilledCircle	draw_filled_circle	Draws a circle filled with specified color
drawFilledEllipse	draw_filled_ellipse	Draws an ellipse filled with specified color
drawLine	draw_line	Draws a line between the given coordinates
drawRect	draw_rect	Draws a rectangle
displayFormat	display_format	Makes a copy of itself on a new surface; used for fast blitting
displayFormatAlpha	display_format_alpha	As displayFormat wtih alpha value per pixel
fillRect	fill_rect	Fills given rectangle with specified color
flags		Returns surface flags
format		Returns pixel format
getClipRect	get_clip_rect	Returns clipping rectangle for the given surface
getPalette	get_palette	Returns the palette of the specified surface
GetPixel	get_pixel	Gets color of the specified pixel
GetRGBget_rgb		Returns RGB component values of specified pixel in an array
getRGBA	get_rgba	Like getRGB, but includes an alpha value
h		Return height
load		Loads image (such as a BMP) and returns instance of SDL::Screen
loadBMP	load_bmp	Loads given bitmap
lock		Sets up a surface for directly accessing pixels
makeCollisionMap		Creates a collision map
mapRGB	map_rgb	Maps the RGB color value to the pixel format of specified surface and

Table 10.2. Common SDL Surface Methods

Method	Equivalent To	Purpose
		returns the pixel value as an integer
`mapRGBA`	`map_rgba`	Same as `MapRGB` but also includes an alpha value
`mustLock?`	`must_lock?`	Returns true if surface must be locked to directly access pixels
`put`		Draw given image in `self`
`PutPixel`	`put_pixel`	Writes pixel to the specified position
`rotateScaled Surface`	`rotate_ scaled_surface`	Rotates surface instance with given angle and scale. Note: method is considered obsolete; it's been superceded by `transformSurface`.
`rotateSurface`	`rotate_surface`	As `rotateScaledSurface` but scale is set to 1.0
`saveBMP`	`save_bmp`	Saves file in BMP format
`setAlpha`	`set_alpha`	Used to set alpha and per-pixel alpha blending
`setColorKey`	`set_color_key`	Sets the colorkey of a blit-able surface
`setColors`	`set_colors`	Same as `setPalette` but with different flags
`setPalette`	`set_palette`	Sets a portion of the palette for the given 8-bit surface
`transformSurface`	`transform_surface`	Creates a rotated and scaled image of given surface
`unlock`		Unlocks a surface
`w`		Returns width

Events

Ruby SDL has two event classes, `event`and `event2`, for handling events. Each has a number of methods; these methods are outlined in Tables 10.3 and 10.4.

Table 10.3. Event Methods

Method	Equivalent To	Purpose
`appState`	`Event.app_state`	Returns current `stat.enableUNICODE`

Table 10.3. Event Methods

Method	Equivalent To	Purpose
`Event.enable_unicode`	`Enable UNICODE`	Keyboard translation (disabled by default)
`Event.disableUNICODE`	`Event.disable_unicode`	Disables Unicode keyboard translation
`Event.enableUNICODE?`	`Event.enable_unicode?`	Returns whether Unicode keyboard translation is enabled
`gain?`		Returns true when gaining focus
`info`		Returns event information in an array
`keyMod`	`key_mod`	Returns the current key modifiers
`keyPress?`	`key_press?`	Returns true when a key is pressed down in a key event
`keySym`	`key_sym`	Returns SDL virtual `keysym`
`mouseButton`	`mouse_button`	Returns the mouse button index
`mousePress?`	`mouse_press?`	Returns true during a mouse button down event
`mouseX`	`mouse_x`	Returns the x coordinate of the mouse
`mouseXrel`	`mouse_xrel`	Returns the relative mouse motion on the x-axis
`mouseY`	`mouse_y`	Returns the y coordinate of the mouse
`mouseYrel`	`mouse_yrel`	Returns the relative mouse motion on the y-axis
`new`		Creates a new `SDL::Event` object
`poll`		Polls for currently pending events
`type`		Returns the type of a given stored event
`wait`		Waits for the next available event
`appState`	`app_state`	Returns the kind of `ActiveEven`

Table 10.4. `Event2` Methods

Method	Equivalent To	Purpose
`Active`		Event that occurs when mouse/keyboard focus gains/loss

Table 10.4. `Event2` Methods

Method	Equivalent To	Purpose
`appState`	`Event2.app_state`	Same as `Event.appState`
`enableUNICODE`	`enable_unicode`	Same as `Event.enableUNICODE`
`enableUNICODE?`	`Event2.enable_unicode?`	Same as `Event.enableUNICODE?`
`disableUNICODE`	`disable_unicode`	Same as `Event.disableUNICODE`
`JoyAxis`		Event that occurs when axis of joystick is moved
`JoyBall`		Event that occurs when a joystick trackball moves
`JoyButtonDown`		Event that occurs when joystick button is pressed
`JoyButtonUp`		Event that occurs when joystick button is released
`JoyHat`		Event that occurs when joystick hat moves
`KeyDown`		Event that occurs when a key is pressed
`KeyUp`		Event that occurs when a key is released
`MouseButtonDown`		Event that occurs when a mouse button is pressed
`MouseButtonUp`		Event that occurs when a mouse button is pressed
`MouseMotion`		Event that occurs when the mouse is moved
`poll`		Same as `Event.poll`
`quit`		Event that occurs when a `quit` or `exit` is requested
`SysWM`		Event that occurs when plaform-dependent window manager occurs
`VideoResize`		Event that occurs when windows are resized
`wait`		Same as `Event.wait`

Ruby SDL also has mouse and key classes and methods for mouse and keyboard events; these are outlined in Table 10.5.

Table 10.5. Mouse and Keyboard Events

Method	Equivalent To	Purpose
`Key.disableKeyRepeat`	`Key.disable_key_repeat`	Disables key repeat
`Key.enableKeyRepeat`	`Key.enable_key_repeat`	Sets keyboard repeat rate
`Key.getKeyName`	`Key.get_key_name`	Returns the string of key name
`Key.modState`	`Key.mod_state`	Returns the current of the modifier keys
`Key.press?`		Return true if given key is pressed
`Key.scan`		Scans key state
`Mouse.hide`		Hides mouse cursor
`Mouse.setCursor`	`Mouse.set_cursor`	Used to change the mouse cursor
`Mouse.show`		Shows a mouse cursor
`Mouse.state`		Returns mouse state in array
`Mouse.warp`		Sets the position of the mouse cursor

Audio

Ruby's SDL has a `Mixer`module that is used to serve up music files, change volume, and set up sound effects like fading. `Mixer`has a class for handling WAV files, `SDL::Mixer::Wave`, and a class for loading music, `SDL::Mixer::Music. Wave` handles standard WAV files, while `Music` can load mod, S3M, it, XM, MID, and MP3 file formats. `Mixer`'s methods are outlined in Table 10.6.

Table 10.6. `Mixer` Methods

Method	Equivalent To	Purpose
`allocateChannels`	`allocate_channels`	Dynamically change the number of channels managed by the mixer
`fadeInMusic`	`fade_in_music`	Fade in the given music in milliseconds
`fadeOutMusic`	`fade_out_music`	Fade out the given music in milliseconds
`halt N/A`		Halt playing of a particular channel
`haltMusic`	`halt_music`	Halt music
`load`		Load a music file and return the object of `Mixer::Music`
`open`		Initialize `SDL_mixer`
`play?`		Return whether specific channel is playing

Table 10.6. `Mixer` Methods

Method	Equivalent To	Purpose
		or not
`playChannel`	`play_channel`	Play a WAV on a specific channel
`playMusic`	`play_music`	Play music
`playMusic?`	`play_music?`	Return whether the music is playing
`pause`		Pause on a particular channel
`pause?`		Return whether a particular channel is paused
`pauseMusic`	`pause_music`	Pause music
`pauseMusic?`	`pause_music?`	Return whether the music is paused
`resume`		Resume a particular channel
`resumeMusic`	`resume_music`	Resume music
`rewindMusic`	`rewind_music`	Rewind music
`setVolume`	`set_volume`	Set the volume
`setVolumeMusic`	`set_volume_music`	Set volume
`spec`		Return the audio `spec` in array

Time

SDL uses the notion of ticks to keep track of time. The `getTicks/get_ticks` method will get the number of milliseconds that have passed since SDL was initialized. There is also a `delay` method that will wait a given number of milliseconds before returning; it is used to process scheduled jobs and events.

Japanese Input

Ruby's SDL comes equipped with an SSK module for encoding the Japanese character set. This module relies on the SDLSSK library, and can set the encoding to the Japanese character system (EUCJP), the ASCII-preserving Unicode system (UTF8), or the Shift-JIS Japanese system (SJIS). SSK has a handful of methods; these are outlined in Table 10.7.

Table 10.7. SSK Methods

Method	Purpose
`Context`	Super class that represents the state of input
`Dictionary`	Super class for manipulating user dictionaries
`encoding`	Returns encoding

Table 10.7. SSK Methods

Method	Purpose
`EUCJP`	Sets encoding to EUCJP
`Keybind`	Represents the keybind in SDLSKK input system
`RomKanaRuleTable`	Represents the rule of conversion from Alphabet to Japanese kana
`SJIS`	Sets encoding to SJIS
`UTF8`	Sets encoding to UTF8

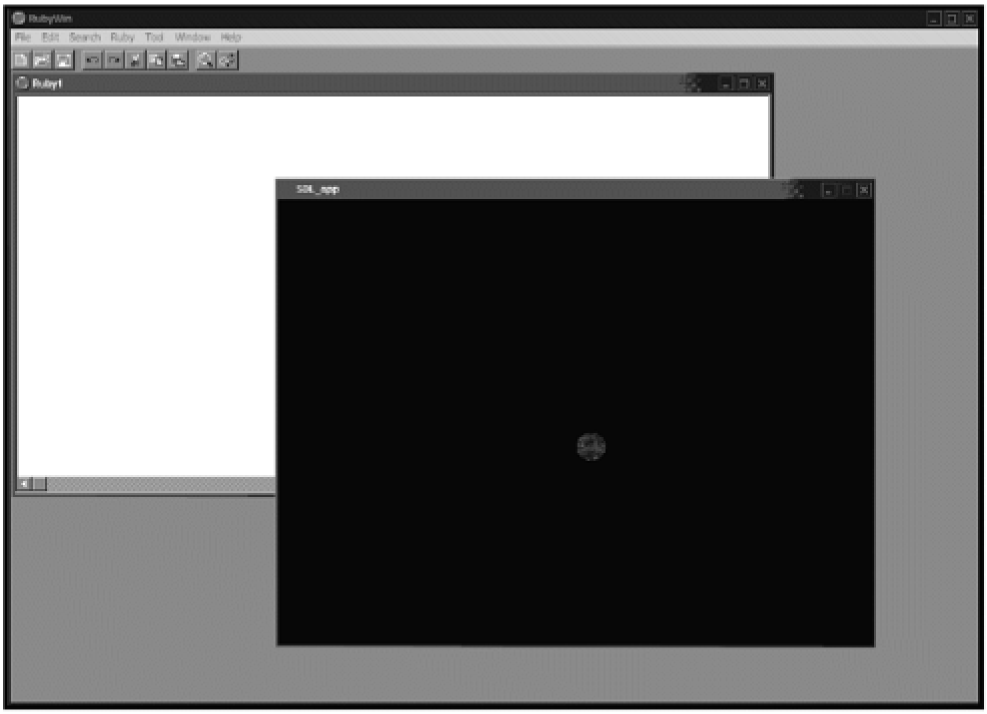
A Sample Ruby SDL Program

All of the tables given in this chapter aren't enough—you need to try an example of using SDL and Ruby together. In the Chapter 10 section of the accompanying CD is a sample RubyBounce folder with five Ruby files. They are as follows:

- CONST.RB
- PLAYER.RB
- RUBYBOUNCE.RB
- STATE.RB
- SYSTEM.RB

These five files are explained in the next few subsections. Each has a part to play in setting up a quick SDL Ruby environment where a player manipulates a small bouncing ruby (see Figure 10.4).

Figure 10.4. A bouncing ruby is displayed in the RubyBounce program.

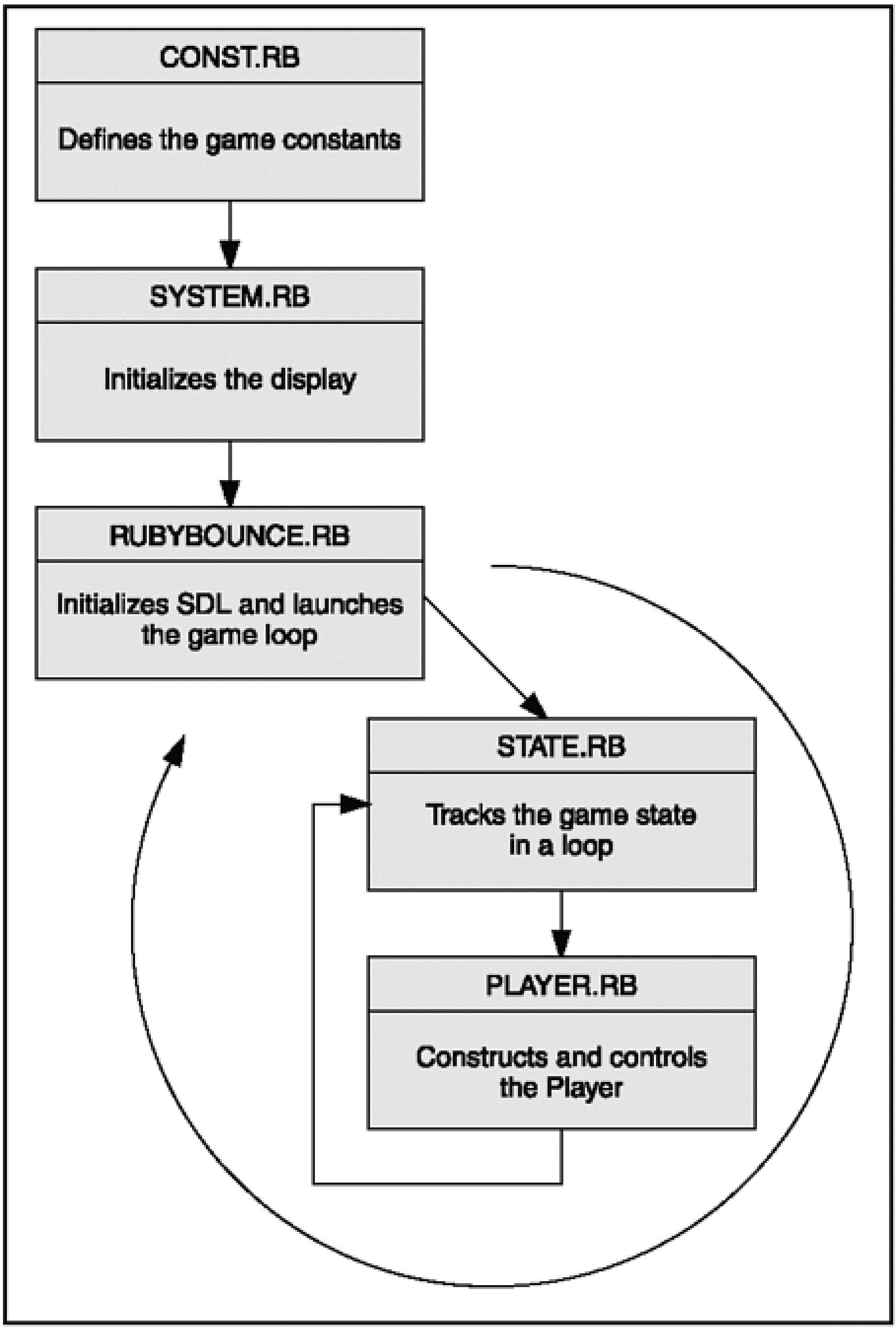


This program can be run from the RubyWin application. Open up rubywin.exe in your new C:\RUBYSDL\BIN folder, choose the Ruby menu, select Run, and then choose the RUBY-BOUNCE.RB file.

The CONST.RB File

The simplest of the five Ruby files, CONST.RB is used to hold any specific game constants that need to be defined (see Figure 10.5). In this example, the file holds four constants, each of which defines a wall in the playing surface. Changing these values later on changes where the player can travel onscreen:

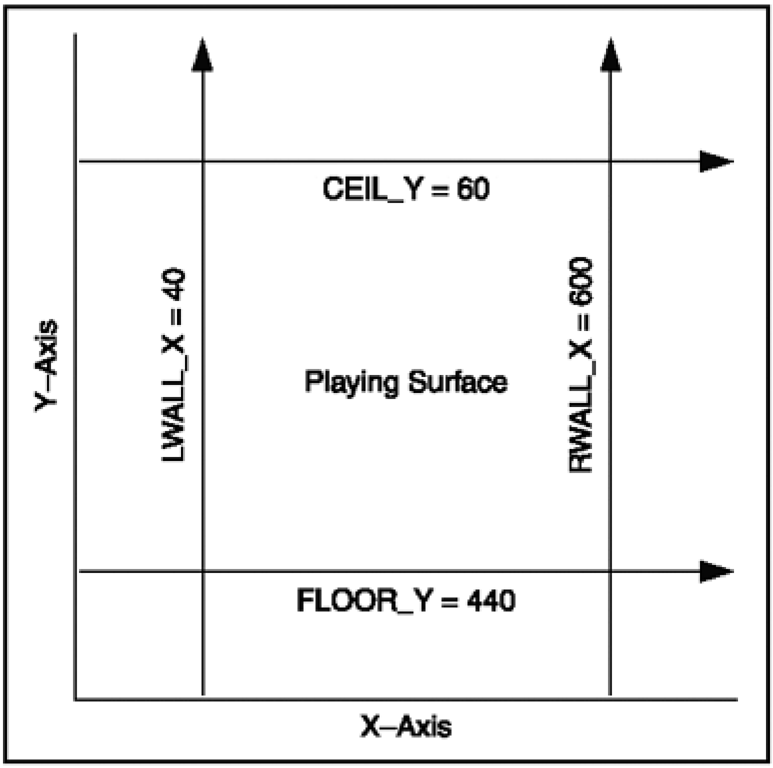
Figure 10.5. File relationship for RubyBounce




```
LWALL_X=40
RWALL_X=600
FLOOR_Y=440
CEIL_Y=60
```

These values are x- and y-set pixel ranges that define the edges of the playing surface in pixels (see Figure 10.6).

Figure 10.6. Playing field x and y boundaries.



The SYSTEM.RB File

The functions set up in the SYSTEM.RB file should look familiar, as they are similar to the functions you used in earlier chapters. The only difference between the first `define`, `setup_bmp`, and earlier endeavors to load bitmaps is Ruby's own unique twist:

```
def setup_bmp(filename)
  graph=SDL::Surface.loadBMP(filename)
  graph.setColorKey SDL::SRCCOLORKEY, graph[0,0]
  graph=graph.displayFormat
end
```

Here `SDL::Surface.loadBMP` is used to grab a .BMP file, the colorkey is set with the `setColorKey` method, and finally, `displayFormat` is used to display the surface.

Also included in this file are two functions for keeping track of where an object travels in the two-dimensional screen. The `x_out` function and the `send_loc` function help determine if the object tries to travel past the `LWALL` and `RWALL` constants set in CONST.RB:

```
def x_out?(x,w)
  x+w+10<LWALL_X || x-10>RWALL_X
end

def send_loc?(x,w)
  return true if LWALL_X+SEND_FIELD_WIDTH>x+w
  return true if RWALL_X-SEND_FIELD_WIDTH<x
  false
```

```
end
```

Then you define the system class with the `initialize` and `continue_game` methods. In a full version game, this would be a good place to set important variables like player score and number of lives, but in this case just one instance variable is set; `@life`:

```
class System

  def initialize
    @life=3
  end

  def continue_game?
    @life > 0
  end

end
```

The STATE.RB File

There are three classes defined in STATE.RB: `State`, `StateInitializer`, and `StateDriver`. Each is used to keep track of the game state, and each is stored within the `jt` (just in time) module. The `State` class has two methods, `initialize` and `move_state`. `State.initialize` is probably the most important method in this script. It first calls the constructor and sets three important instance variables: `state_hash`, `state_driver`, and `state`. Using each variable, `State.initialize` then sets a loop that iterates over each entry in `state_hash`:

```
class State
def initialize(first_state)
state_initializer = StateInitializer.new
yield state_initializer

@state_hash = state_initializer.state_hash
@state_driver = StateDriver.new(self)
@state = first_state

@state_hash.each do |key,val|
self.instance_eval <<-EOS
def self.#{key.id2name}(*arg)
if @state_hash[:#{key.id2name}][@state] then
@state_hash[:#{key.id2name}][@state].call(@state_driver,*arg)
end
end
EOS
end
end
```

The `move_state` method is used to create new states and assign them to `@state`:

```
def move_state(new_state)
@state=new_state
end
```

The `StateInitializer` class defines both `initialize`—which creates the `state_hash` instance variable—and `add_event`:

```
class StateInitializer
def initialize
@state_hash={}
end
attr_reader :state_hash
def add_event(state,event,&block)
                if not @state_hash[event] then
@state_hash[event]={}
end
        @state_hash[event][state]=block
end
end
```

Finally, define the class `StateDriver` with two methods, `initialize` and `move_state`:

```
class StateDriver
def initialize(state_obj)
@state_obj=state_obj
end
def move_state(new_state)
@state_obj.move_state(new_state)
end
end
```

The PLAYER.RB File

Now the fun stuff—the player must be defined with a constructor method (`initialize`). You need methods to display the player onscreen (`w`, `h`, and `draw`) and move around the screen (`act` and `move_lr`). But first, the PLAYER.RB file needs help from SYSTEM.RB and STATE.RB:

```
require 'system.rb'
require 'state.rb'
```

Next, designate the class `Player`and define a few player constants:

```
class Player
  INIT_DY=-50
  DX=20
  H=32;W=32
  G=20
  GRAPH_P1 = setup_bmp 'ruby.bmp'
```

These constants initialize the height and width and name of the bitmap image of the player piece. After that, call the `initialize` method. This method not only calls the

SYSTEM.RB code but it also establishes keyboard events for moving the player's ruby piece around the screen, including moving left and right and jumping the piece up:

```
def initialize(system)
    @system=system
    @x=320;@y=200
    @dy=0
    @state=JT::State.new(:jumping) do |i|
      i.add_event(:walking,:act) do |d,key,dt|
       move_lr(key,dt)
       if key.jump then
         @dy= INIT_DY
         d.move_state :jumping
       end
      end
```

The player pieces must also track the constants set in CONST.RB so that the piece cannot leave the playing field:

```
i.add_event(:jumping,:act) do |d,key,dt|
move_lr(key,dt)
@y += @dy*dt/100
        @dy += G*dt/100
        if @y > FLOOR_Y - H then
@y = FLOOR_Y - H
d.move_state :walking
end
end
```

Included in the `Player.initialize` method are sample event handlers to track the player piece in case it collides with any other sprites/rects on the playing surface:

```
@damage_state = JT::State.new(:normal) do |i|
i.add_event(:normal,:act) { }
i.add_event(:normal,:collision_enemy) do |d|
@system.collision_enemy
@damage_time=0
d.move_state(:damaged)
end
i.add_event(:damaged,:act) do |d,dt|
@damage_time+=dt
d.move_state(:normal) if @damage_time > DAMAGE_TIME
end
i.add_event(:damaged,:collision_enemy) { }
end
end
```

After `Player.initialize` come two quick methods that define the width and height of the player piece:

```
def w ;W;end;
def h ;H;end;
```

You need a `draw` method to put the previously defined bitmap (in `GRAPH_P1`) onto the screen. Drawing the bitmap is accomplished with the `put` method:

```
def draw(screen)
screen.put(GRAPH_P1,@x,@y)
end
```

The `act` method is a worker method that checks with STATE.RB and establishes the `state.act` and `damage_state.act` instance variables so that the player piece has the functionality from STATE.RB:

```
def act(key,dt)
@state.act(key,dt)
@damage_state.act(dt)
end
```

Finally, define the player's movement within a `move_lr` method. `Move_lr` checks whether the player's key presses move the actual game piece off of the predefined playing surface:

```
def move_lr(key,dt)
@x-=DX*dt/100 if key.left
@x+=DX*dt/100 if key.right
@x = LWALL_X if @x< LWALL_X
@x = RWALL_X-W if @x > RWALL_X-W
end
```

The RUBYBOUNCE.RB File

It's in RUBYBOUNCE.RB that SDL is opened and initialized and the actual game loop runs. First, SDL and the other defined files are required:

```
require 'sdl'
require 'system.rb'
require 'state.rb'
require 'const.rb'
require 'player.rb'
```

Initialize SDL with its `init` method, define the video mode, and establish the surface area with the following two lines:

```
SDL.init( SDL::INIT_VIDEO )
screen = SDL::setVideoMode(640,480,16,SDL::SWSURFACE)
```

A new structure is established that holds each keypress available to the player:

```
Key = Struct.new("Key",:left,:right,:jump,:send)
```

The `new` method constructor is called for each object that must be initialized:

```
system=System.new
player=Player.new(system)
event=SDL::Event.new
key=Key.new
```

Now that every object you need is established, the game loop is created. First, use `tick` to establish the time:

```
before=now=SDL::getTicks-1
```

Then establish a `while` loop that uses the `poll` method to check for events from the keyboard:

```
while system.continue_game?
  if  event.poll != 0 then
    if event.type==SDL::Event::QUIT then
      break
    end
    if event.type==SDL::Event::KEYDOWN then
      exit if event.keySym==SDL::Key::ESCAPE
    end
  end
```

Each possible key press is queried for by `Key::press?`:

```
SDL::Key::scan
key.left = SDL::Key::press?(SDL::Key::LEFT)
key.right = SDL::Key::press?(SDL::Key::RIGHT)
key.jump = SDL::Key::press?(SDL::Key::UP)
key.send = SDL::Key::press?(SDL::Key::DOWN)
```

The SDL ticks are checked for in the loop as time moves forward:

```
before=now
now=SDL::getTicks
dt=now-before
```

Any actions are fulfilled by calling `player.act`:

```
player.act(key,dt)
```

The screen is filled, and the player redrawn with each iteration of the loop:

```
screen.fillRect(0,0,640,480,0)
player.draw(screen)
```

All that is left to do is make sure the SDL screen is flipped and that any garbage is collected:

```
ObjectSpace.garbage_collect
screen.flip
```

Summary

That's a wrap on the common Ruby game libraries. A few important points to take from this chapter are as follows:

- Ruby distributions commonly come equipped with pieces needed to make games and GUIs, and it is common to find FOXRuby and Ruby OpenGL.
- Python, Lua, and Ruby each have tools for using SDL and OpenGL.
- SDL and OpenGL in Ruby look really similar to SDL and OpenGL in other languages.
- There are a number of significant differences to how FOX is implemented in Ruby than in other languages.

Questions and Answers

1: Q: What platforms does RubySDL operate on?

A: A: Linux, Win32, FreeBSD, and BeOS.

2: Q: How do I use threads in RubySDL?

A: A: RubySDL cannot handle SDL threads. However, Ruby's threads can be used instead.

Exercises

1: List three common Ruby SDL surface methods.

2: FXRuby is most commonly used to _____.

3: Use the PLAYER.RB code as a sample to create an OBSTACLE.RB script that can add obstacles to the RubyBounce code.

Chapter 11. The Ruby Game Community

The gem cannot be polished without friction, nor man perfected without trials.

——Chinese proverb

Ruby is probably the least entrenched of the languages within English-speaking game-development companies. It is found much more often in the scientific and research community than in the game and entertainment industries. This doesn't mean that Ruby can't be found hard at work in the game field, though—it's been a part of a number of large-scale projects involving games. This chapter highlights a few Ruby projects associated with games or game-related technologies.

Ruby and Game Engines

Ruby is available as a tool for a few game-programming engines. ClanLib is an engine I've mentioned before that has been used on many independent projects, while MUES is a new Ruby tool known primarily as the backbone of The FærieMUD World.

ClanRuby

ClanLib (www.clanlib.org) is one of the more popular libraries for amateur game designers today. ClanLib is written entirely in C++ as a graphics and game library. It takes the hard-to-develop functionality like sound mixing, DirectDraw, networking, and working with images and provides an easy-to-use, multi-platform library to develop this functionality. ClanLib also provides low-level interfaces to other popular libraries such as DirectFB, DirectX, OpenGL, and X11.

ClanRuby is a set of bindings from Ruby that tie into ClanLib's library. ClanLib and ClanRuby are licensed under the GNU Library General Public License. ClanRuby was developed by Russell Olsen, and, as of this writing, Version 0.6.5a—which is compatible with ClanLib 0.6.5—is available at its Sourceforge project page at http://sourceforge.net/projects/clanruby/. ClanRuby's home page—which shows the latest ClanRuby developments, offers a brief ClanRuby tutorial and shows sample ClanRuby uses—is at http://clanruby.sourceforge.net.

ClanRuby 0.6.5 is also included on this book's CD in the Chapter 11 folder. Russell Olsen has tested the platform primarily on Red Hat Linux 7.3 using ClanLib 0.6.5 and Ruby 1.6.7 or 1.6.5. ClanRuby can be installed from the source by unpacking the tar files, running the Ruby configuration script (EXTCONF.RB), which creates the make file, and then running `make` and then `make install`.

While ClanRuby currently only works in Red Hat, ClanLib delivers a platform-independent interface. If a game is written with ClanLib, it should be possible to compile the game under just about any platform without changing the application source code.

But ClanLib is not just a wrapper library, providing a common interface to low-level libraries such as DirectFB (`Direct Frame Buffer`), DirectX, OpenGL, X11, and so on. While platform independence is ClanLib's primary goal, it also tries to be a service-minded game SDK. In other words, a lot of effort has been put into designing the API in order to ensure that ClanLib is easy to use but still quite powerful.

ClanRuby can be brought into a Ruby program after installation with the following:

```
require 'ClanRuby'
include ClanRuby
```

Setting up the ClanRuby environment is accomplished with a few `init` methods and `Display.setVideoMode`, which set the screen size and resolution. Cleanup is handled by `deinit` methods:

```
#Initialize
SetupCore.init()
SetupDisplay.init()

#Set Display 640x480x16bit
Display.setVideoMode( 640, 480, 16, false )

#
#Actual Bulk of the program here
#

#De-initialize
SetupDisplay.deinit()
SetupCore.deinit()
```

Users of ClanLib will recognize the upcoming code. Those who have delved into OpenGL and SDL in earlier chapters will also find ClanLib's syntax familiar; for instance, here's how to draw a rectangle:

```
Display.fillRect() #Parameters to define where rect is drawn go here
Display.flipDisplay()
```

Sourceforge is also home to a few games written in ClanLib, including a Boulderdash clone called Epiphany written by Guiseppe D'Aqui; it's at htttp://epiphany.sourceforge.net.

MUES

The MUES (Multi-User Environment Server) is a game-environment server written in Ruby. The purpose of MUES is to facilitate building online multiplayer games or simulations. It provides game worlds in the form of dynamically programmed object environments, machine services and daemons for creating in-game systems, and a network client to access these environments.

MUES is just the first half of the project—the programming of the server platform. MUES is also tied into a MMORPG (Massively Multi-player Online Role-Playing Game) called FærieMUD, which is the creative, vision-inspired, story-based world the development team has been building in conjunction with the engine.

MUES itself is open-source software that was released to the public in late 2001. The source code and documentation can be found at http://mues.faeriemud.org/. The MUES engine supports a number of useful MUD features, including:

- Multi threading
- I/O abstraction
- Network sockets and protocols
- Object persistence
- Logging
- Dynamic/data driven environment
- User authentication

Ruby and Graphics

Because Ruby is the new kid on the block, a number of the graphical Ruby projects are still very much under construction. The projects I will present in the next subsections have been around for a while, have proven their usefulness in a number of applications, and can be found packaged with good examples and documentation. In contrast to Python and Lua libraries, Ruby graphic libraries tend to be scientific or Web-based in nature.

FXRuby

I offered a quick look at this toolkit and OpenGL in Chapter 10. In addition to working with OpenGL, FXRuby can also work with Scintilla—or at least with FXScintilla, which is FOX's wrapper around the Scintilla library.

More information on FXRuby can be found at its Website, at http://www.fxruby.org/

Ruby/PGPlot

Ruby/PGPlot is a Ruby interface to the PGPlot graphics library. PGPlot is, itself, a device-independent graphics package specifically designed for plotting graphs of publication quality. PGPlot is not public domain software, but it is available free of cost for non-commercial endeavors. It also has hooks into several other languages, including Ada, C, FORTRAN, and Python. Ruby/PGPLot relies on Numeric Ruby; the technologies and URLs are

- **Ruby/PGPlot.** http://www.ir.isas.ac.jp/~masa/ruby/pgplot/index.html.
- **Numeric Ruby.** http://www.ir.isas.ac.jp/~masa/ruby/index-e.html#pgplot.
- **PGPlot.** http://www.astro.caltech.edu/~tjp/pgplot/.

RubyDCL

RubyDCL is a Ruby interface, written by T. Horinouchi, K. Kuroi, and K. Goto that hooks into the DCL scientific-graphics library. The interface supports all of DCL—every function and subroutine—and, although much of the documentation is in Japanese, it does come with some English-language documents and support. Ruby DCL is part of a larger project, the Dennou Ruby Project, the purpose of which is to develop a suite of software that facilitates visual scientific simulations.

The DCL graphics library was originally written in FORTRAN and later ported into C, and Ruby DCL is actually the second version of the product built by Dennou. The first library, AdvancedDCL, was the experimental prototype for RubyDCL and is now obsolete.

RubyDCL, the Dennou Ruby Project, and the RubyDCL project page can be found online at the following links:

- **RubyDCL.** http://ruby.gfd-dennou.org/products/ruby-dcl/.
- **Dennou Ruby Project.** http://ruby.gfd-dennou.org/.

- **RAA RubyDCL Project Page.** http://raa.ruby-lang.org/list.rhtml?name=rubydcl.

Libgd-Ruby

GD is a library by Thomas Boutell that is used for dynamically creating graphic images, particularly PNG and JPEG images, and Libgd-Ruby is a package extension library that allows Ruby to wrap around GD. Libgd is written in C, and is considered freeware. It is dependent on several other libraries, including libc6, the GNU C Library, FreeType 2, the GD Graphics Library, and The Independent JPEG Group's JPEG runtime library. Details can be found at the following links:

- **Libgd-Ruby.** http://packages.debian.org/unstable/graphics/libgd-ruby.html.
- **GD Library.** http://www.boutell.com/gd/faq.html.

Ruby and Games

Ruby is the newcomer in the American game industry, but gamers can expect many good things to come. Ruby is perfect for Internet-based games such as FærieMUD, which is highlighted below.

The FærieMUD Project

Built in tandem and integrated with MUES, The FærieMUD project is built to be story-rich, with a focus on detail, realism, and imagination—unlike the all-too-common violent fantasy world.

FærieMUD was originally written in Perl and was ported over to Ruby to take advantage of a few Ruby features like built-in meta classes, strict encapsulation, and pure OOP.

The FærieMUD Project is still being built, and can be found at http://www.faeriemud.org/.

The MUES engine can be found online at http://mues.faeriemud.org/.

Beyond Ruby

Ruby is the shining star in a few other domains besides game graphics and game engines. A few of these are listed in the following sections.

The Snack Sound Toolkit for Ruby

The Snack Sound Toolkit is a collection of sound- and voice-processing routines and includes tools for speech recognition, formant tracking and synthesis, and other fun sound-and speech-based tools. The toolkit, written by Stephen Legrand, is used to extend scripting languages and enable such tools within them; Ruby is its prodigy pilot-child.

The original implementation of Snack was inspired by Kåre Sjölander and was extended to Tcl/Tk. Snack for Ruby leverages the existing Tk graphics and provides direct support for waveforms and spectrograms.

Snack for Ruby requires that the Snack package and Tcl/Tk be installed. RPMs that combine both Snack and Tcl/TK in one install are available for Linux users, and the toolkit runs on both Posix and Windows environments.

Snack for Ruby is still under development but was presented at the International Ruby Conference.

The toolkit can be found on Sourceforge at http://sourceforge.net/projects/rbsnack/.

rbwrap

rbwrap is a tool for converting Ruby scripts and programs into standalone executables. The tool is in Alpha currently but works for Windows systems. The package relies on Cygwin and the Gnu C Compiler.

rbwrap is written by Robert Feldt and can be found at the author's Website, at http://www.ce.chalmers.se/~feldt/ruby/applications/rbwrap.

Memeoize

Memeoize is a tool for speeding up program execution. It does so by caching functions, increasing the size of the running program but also speeding up execution time. Memoize is also the brainchild of Robert Feldt; it relies on Cygwin and is meant to work with Ruby 1.6.2.

Memeoize is available online at http://www.ce.chalmers.se/~feldt/ruby/extensions/memeoize.

Summary

That's a wrap on Ruby. I hope you enjoyed your stay. A few important points to take from this chapter:

- Even though Ruby tools appear more frequently and are used more often in the scientific world than in the entertainment and game industries, they can still be found if you look hard enough.
- Ruby has quite a bit of support for developing scientific graphs and charts. Integration with some of the libraries that allow the rapid development of Ruby GUIs makes Ruby a good tool for a number of research facilities.

Questions and Answers

1: Q: Where can I find more Ruby projects?

A: A: The Ruby community updates a Web page with active Ruby projects at the Ruby Application Archive at http://raa.ruby-lang.org/.

2: Q: Have there been many games written with Clanlib?

A: A: Clanlib has been involved in dozens of games, and most of them are listed on the Clanlib Web site at http://www.clanlib.org/games.html/.

Exercises

1: List three available resources for programming games in Ruby.

2: Finish this statement: "The Snack toolkit deals mostly with _____."

Part FIVE: The Wrap Up

The book wrap-up discusses taking what you've learned so far into other areas. The main topics are using extension as a technique in development and wrapping high-level languages into C.

Chapter 12. Using Python, Ruby and Lua in Development

The game is up.

——William Shakespeare, Cymbeline

High-level languages are capable of working with other programming tools. Discussed in this chapter are common ways these languages can be brought in to work as part of a team. I'll cover, with examples, extension and wrapping, as well as integrating the languages with C.

High-Level Languages in the Development Cycle

There are a number of advantages for using a high-level language in a development project. These advantages include

- Automated garbage collection.
- High-level features like built-in pattern matching and built-in types.
- Simpler syntactical rules.
- Coding is less time-consuming.
- Lower costs than using an internally built language.
- High-level languages are easily embedded, modular, and extensible.
- Artists, level designers, and even employees with little computer science experience can easily grasp and understand high-level languages.

There are also a number of reasons to not use a high-level language for a development project. These include

- They are slower.
- Their byte-code can be easier to hack.
- Their debuggers aren't as advanced.
- Legal concerns could arise when using open-source code in for-profit development.

The key, then, is to know when to use the tools and when not to use the tools. Although Python, Lua, and Ruby can be used to write complete games, they usually aren't. In a typical shop they serve a specific function, where their strengths can be leveraged.

For instance, in a Python game, the main looping engine code may look something like the following:

```
# Update any input from the User
Input.GetInput()
# Process user Input
Input.ProcessInput()
# Use tick to up-date the graphics scene
Graphics.Tick()
#Redraw the graphics
Graphics.Redraw()
```

There is nothing that says each of the calls must be Python, however. Python can be calling to modules written in other languages. The `Graphics.Tick` and `Redraw` methods could be ANSI C or even assembly. Python could be running the game loop and calling out to C only when needed for CPU-intensive operations.

In a project that mixes languages, you'll likely see two languages, as shown in Figure 12.1. One will be high-level, used for generic tasks, and administrative. The low-level language is used for specific time-saving tasks (see Table 12.1).

Figure 12.1. Typical roles of partnered languages

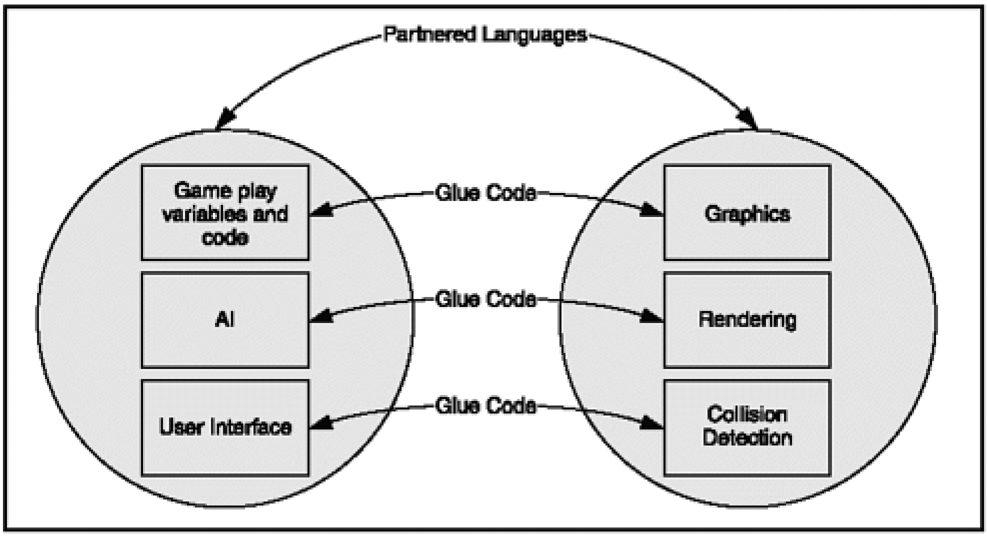


Table 12.1. Partnered Languages

Typical High-Level Language Tasks	Typical Low-Level Language Tasks
Call low-level language	CPU intensive tasks
Game code default	Graphics/rendering system
AI	Collision detection
User interface	Tasks with many quick iterations

Perhaps the biggest benefit to development is using a scripting language to drive data. Over the years, companies have discovered that it is not a good idea to bury game parameters— like movement speed, character strength, and unit hit points for example—deep down in executable code. If these attributes are buried, play testing becomes an extremely lengthy process because every little change must be made to complex, difficult-to-read-and-understand code, and then there must be a lengthy recompilation and re-building of the entire game. Rebuilding a game with a large code base from scratch can take hours, or even a whole day, and the act of recompiling actually risks introducing new bugs or issues.

If game parameters (like movement speed, character strength, and unit hit points) can instead be controlled with a scripting language, they can be changed almost on-the-fly. Play testers could change statistics and attributes until the balance of the game makes sense without having to go back to a development team.

Also, game play details can be really time-consuming to program in C or C++. If high-level scripting is running the AI, the player attributes, or the quest flow of the game, then the C coder will be freed up to focus on the engine code. Designers can easily fiddle with the settings of the higher-level code and try out parameters that they normally would have to delve deep into the engine to get to.

Even better, if the separation between the engine and the game code is severe enough, the base low-level engine can actually be used for multiple games and multiple releases. The C/C++ engine stays static while the high-level scripts define new parameters, new game objects, and new goals and missions for the new player characters. Since many companies claim that the biggest problem they face is resource management, you can see why many have adopted this release philosophy.

Extending Python, Lua, and Ruby

Extending is one of the super powers Python, Lua, and Ruby have to offer. Extending is basically the ability to combine code from two or more different languages into one running executable or script. Although this adds a layer of complexity to a project, it gives a developer the ability to pick and choose from the existing toolbox.

All of these languages are built around being extensible; extensibility is one of the features that has made them so prolific. The language documentation that comes with each includes a nifty sample and explanation of how to partner with other languages, so this section is more of a brief overview of the process.

Languages are extended for many different reasons. A developer may want to use an existing C library or port work from an old project into a new development effort. Often extensible languages are used as prototypes, and then profiling tools are used to see what parts of the code execute slowly, and where pieces should be re-written. Sometimes a developer will need to do something that just isn't possible in the main language, and must turn to other avenues.

Extending is mainly used when another language can do the job better—better meaning more efficiently or more easily. Most commonly, you will find these languages partnered with C and C++, where the Cs are running code that needs to be optimized for speed and memory.

Problems with Extending

As I've already mentioned, multilanguage development adds an extra layer of complexity. Particular problems with extending are as follows:

- You must debug in two languages simultaneously.
- You must develop and maintain glue code that ties the languages together (this might be significantly large amounts of code).
- Different languages may have different execution models.
- Object layouts between languages may be completely different.
- Changes to one side of the code affect the other side, creating dependencies.
- Functions between languages may be implemented differently.

Extended programs can also be difficult to debug. For instance, Ruby uses the GNU debugger, which can look at core dumps but still doesn't have breakpoints or access to variables or online source help. This is really different from the types of tools available for C and C++, where breakpoints and core dumps can be watched and managed during debug execution. Since the tools can differ between two languages, a developer may have to hunt through more than one debugger to find a problem. Also, because high-level language debuggers are usually more primitive, there is less checking during compile time, which could lead to missed code deficiencies.

There are some glue code packages that solve some of these problems. These are third-party programs that manage the creation of extended code; Simple Wrapper Interface Generator (SWIG, covered later in the chapter) is one example of such a package.

Though adding more than one language to a project gives you more options, as I said, it does add an extra level of complexity. When you add a language, you will need multiple compliers and multiple debuggers, and you will have to develop and maintain the glue code between the two languages. Whether to add a language is a tough management question, one that needs to be answered based on the needs of each particular project.

A final issue with having high-level code in a shipped product is that the code reveals much more about the source than does C or C++; this can make it more vulnerable to hacking. This doesn't mean that C or C++ cannot be hacked, just that if the variable names and function names are shipped in scripts with the game code in a high-level format, the game can be easier to break into or deconstruct.

Extending Python

There are a few built-in ways of integrating Python with C, C++, and other languages. Writing an extension involves creating a wrapper for C that Python imports, builds, and can then execute. Python also provides mechanisms for embedding, which is where C (or an equivalent) is given direct access to the Python interpreter. There are also a number of third-party integration solutions.

Writing a Python Extension

You must write a wrapper in order to access a second language via a Python extension. The wrapper acts as glue between the two languages, converting function arguments from Python into the second language and then returning results to Python in a way that Python can understand. For example, say you have a simple C function called `function`:

```
int function (int x)
        {
/*code that does something useful*/
        }
```

A Python wrapper for `function` looks something like the following:

```
#include <Python.h>
PyObject *wrap_function(PyObject *self, PyObject *args)
{
                int x, result;
                if (!PyArg_ParseTuple(args, "i:function",&x))
                        return NULL;
                result = function(x);
                return Py_BuildValue("i",result);
        }
```

The wrapper starts by including the Python.h header, which includes the necessary commands to build a wrapper, and also a few standard header files (like stdio.h, string.h, errno.h, and dstlib.h).

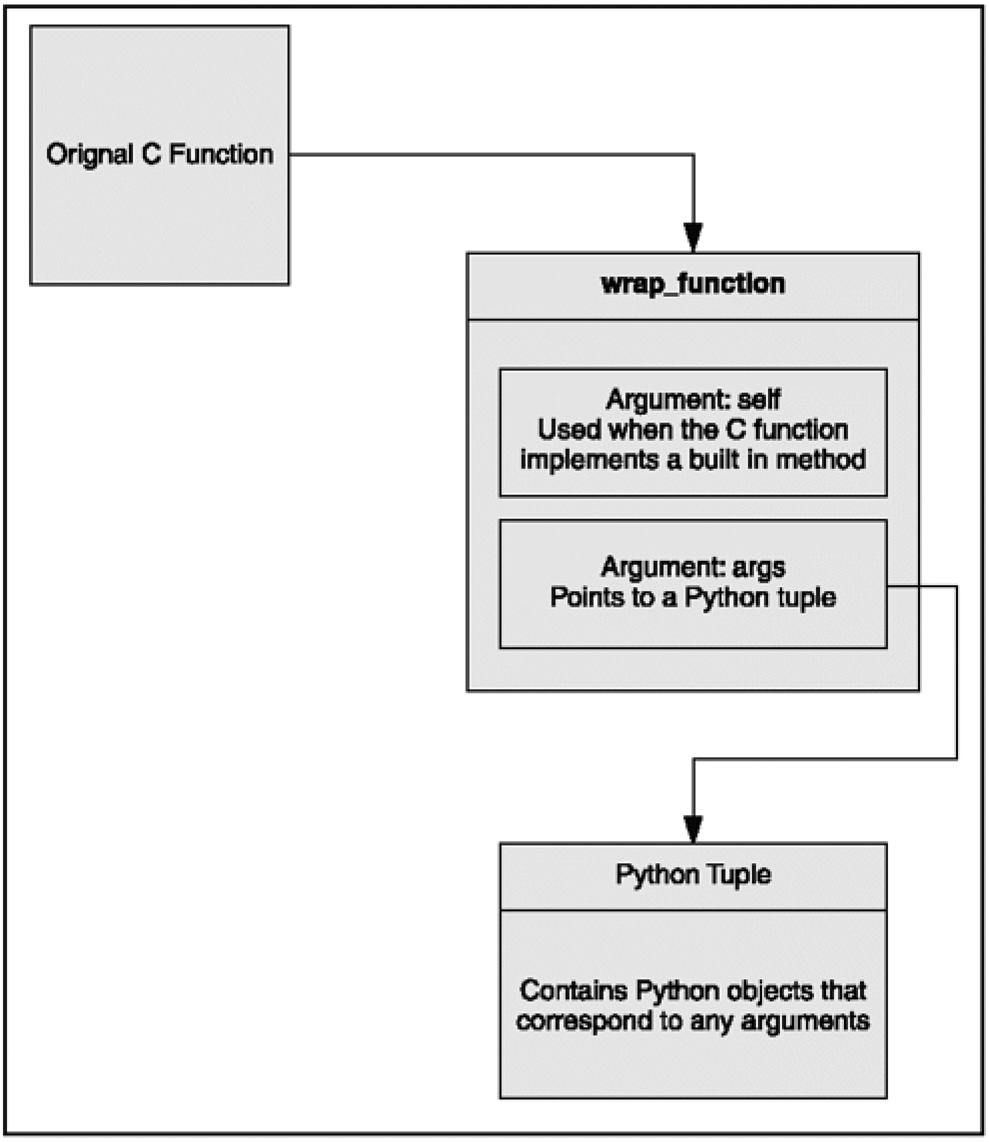
NOTE

TIP

Python commands that are included with Python.h almost always begin with Py or py, so they are easily distinguished from the rest of the C code.

The `PyObject` wrapper `wrap_function` has two arguments, `self` and `args` (see Figure 12.2). The `self` argument is used when the C function implements a built-in method. The `args`argument becomes a pointer to a Python tuple object containing the arguments. Each item of the tuple is a Python object and corresponds to an argument in the call's argument list.

Figure 12.2. The illustrated `wrap_function`

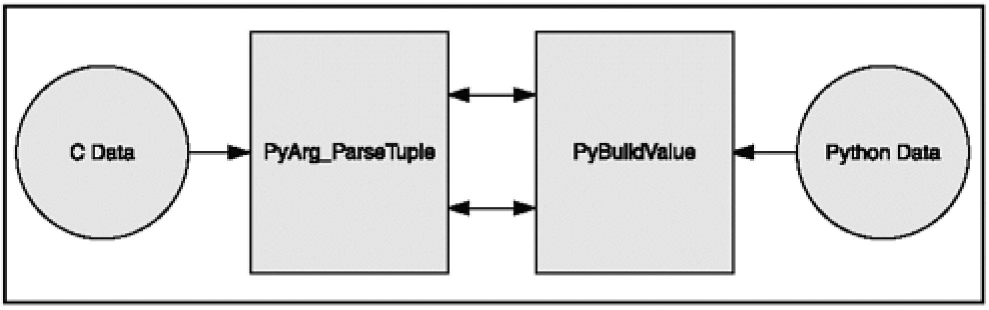


The small "i" in the `i:function`line is short for int. If the function instead required a different type, you would need to use a different letter than "i":

- **i.** For an integer.
- **I.** For a long integer.
- **s.** For a character string.
- **c.** For a single character.
- **f.** For a floating point number
- **d.** For double
- **o.** For an object
- **Tuple.** Python tuples can hold multiple objects.

Together, `PyArg_ParseTuple()` and `PyBuildValue()` are what converts data between C and Python (see Figure 12.3). Arguments are retrieved with `PyArg_ParseTuple`, and results are passed back with `Py_BuildValue`. `Py_BuildValue()` returns any values as Python objects.

Figure 12.3. Data converting between C and Python



`PyArg_ParseTuple()` is a Python API function that checks the argument types and converts them into C values so that they can be used. It returns true if all arguments have the right type and the components have been stored in the variables whose addresses are passed. If a C function returns no useful argument (i.e. `void`), then the Python function must return `None`.

In the code snippet an `if`statement is also used. This structure is there just in case an error is detected in the argument list. If an error is detected, then the wrapper returns `NULL`.

Once a wrapper has been written, Python needs to know about it. Telling Python about the wrapper is accomplished with an initialization function. The initialization function registers new methods with the Python interpreter and looks like this:

```
Static PyMethod exampleMethods[] = {
        {"function", wrap_function, 1},
        {NULL, NULL}
};

void initialize_function(){
        PyObject *m
        m = Py_InitModule("example", "exampleMethods");
}
```

Only after a wrapper and an initialization function exist can the code compile. After compilation, the function is part of Python's library directory and can be called at any time, just like a native Python module.

You can also use a setup file when importing a module. A setup file includes a module name, the location of the C code, and any compile tags needed. The setup file is then pre-processed into a project file or makefile.

The compile and build process for extending varies, depending upon your platform, environment, tools, and dynamic/static decision-making, which makes the Python parent documentation extremely valuable when you're attempting this sort of development.

Guido Van Rossum has a tutorial on extending and embedding Python within the language documentation, at http://www.python.org/doc/current/ext/ext.html.

The Python C API Reference manual is also extremely helpful if C or C++ is your target language. It's at http://www.python.org/dev/doc/maint22/api/api.html.

The last step in Python extension is to include any wrapped functions (in this case, `function`) in the Python code. Do this with a simple import line to initialize the module, like so:

```
import ModuleToImport
```

Then the function can be called from Python just like any other method.

```
ModuleToImport.function(int)
```

Embedding Python

Embedding in Python is where a program is given direct access to the Python interpreter, allowing the program the power to load and execute Python scripts and services. This gives a programmer the power to load Python modules, call Python functions, and access Python objects, all from his or her favorite language of comfort.

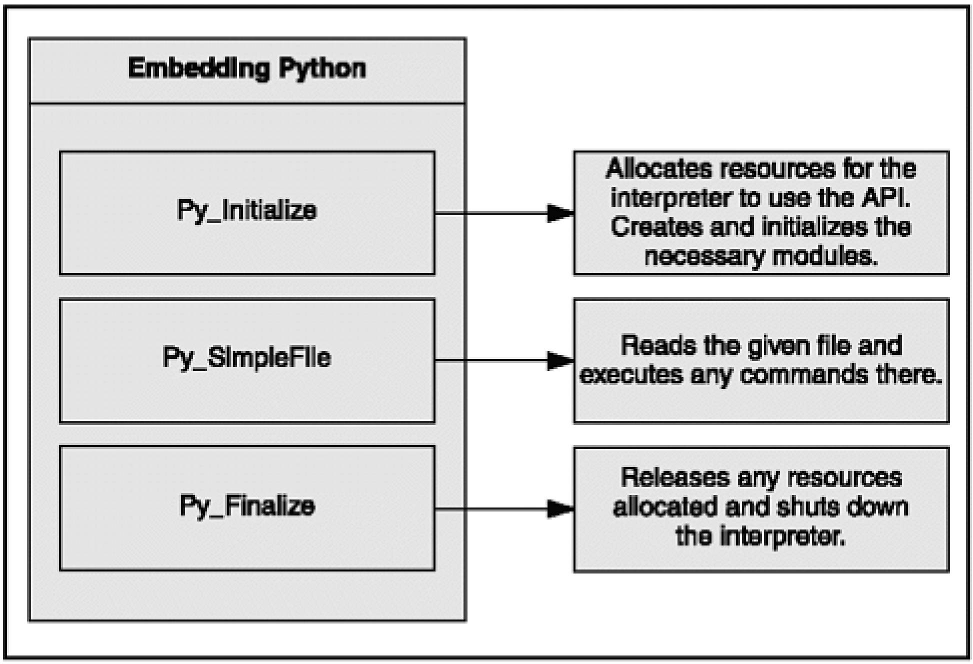
Embedding is powered by Python's API, which can be used in C by including the `Python.h` header file. This header

```
#include "Python.h"
```

contains all the functions, types, and macro definitions needed to use the API.

It is fairly simple to initialize Python in C once the Python header file is included (see Figure 12.4):

Figure 12.4. The embedding Python process




```
int  main()
        {
                Py_Initialize();
                PyRun_SimpleFile("<filename>");
                Py_Finalize();
                return();
        }
```

`Py_Initialize` is the basic initialization function; it allocates resources for the interpreter to start using the API. In particular, it initializes and creates the Python `sys`, `exceptions`, `_builtin_`, and `_main_modules`.

NOTE

CAUTION

`Py_Initialize` searches for modules assuming that the Python library is in a fixed location, which is a detail that may need to be altered, depending on the operating system. Trouble with this function may indicate a need to set the operating system's environment variable paths for `PYTHONHOME` or `PYTHON PATH`. Alternately, the module paths can be explicitly set using `PySys_SetArgv()`.

The `Pyrun_SimpleFile`function is simply one of the very high-level API functions that reads the given file from a pointer (`FILE *`) and executes the commands stored there. After initialization and running any code, `Py_Finalize` releases the internal resources and shuts down the interpreter.

Python's high-level API functions are basically just used for executing given Python source, not for interacting with it in any significant way. Other high-level functions in Python's C API include the following:

- **Py_CompileString().** Parses and compiles source code string.
- **Py_eval_input.** Parses and evaluates expressions.
- **Py_file_input.** Parses and evaluates files.
- **Py_Main().** Main program for the standard interpreter.
- **PyParser_SimpleParseString().** Parses Python source code from string.
- **PyParser_SimpleParseFile().** Parses Python source code from file.
- **PyRun_AnyFile().** Returns the result of running `PyRun_InteractiveLoop` or PyRun_SimpleFile().
- **PyRun_SimpleString().** Runs given command string in `_main_`.
- **PyRun_SimpleFile().** As `PyRun_SimpleString` except source code can be read from a file instead of a string.
- **Py_single_input.** Start symbol for a single statement.
- **PyRun_InteractiveOne().** Read and execute a single statement from an interactive device file.
- **PyRun_InteractiveLoop().** Read and execute all statements from an interactive device file.
- **PyRun_String().** Execute source code from a string.
- **PyRun_File().** Execute source code from a file.

The high-level tools really just scratch the surface, and Python's API allows memory management, object creation, threading, and exception handling, to name a few things. Other commonly used commands include `PyImport_ImportModule()`, which is for importing and initializing entire Python modules; `PyObject_GetAttrString()`, which is for accessing a given modules attributes; and `PyObject_SetAttrString()`, which is for assigning values to variables within modules.

Third-Party Integration

So what happens when there is a large integration project and some 100+ C functions must be gift-wrapped for Python? This can be a time-consuming, tedious, error-prone project. Imagine now that the library goes through a major update every four to six months, and each wrapper function will need to be revisited. Now you know what job security looks like!

Luckily, there are other options available for extension besides wrappers. SWIG, for instance, is an extension wrapper designed to make extension easier. It can be used to generate interfaces (primarily in C) without having to write a lot of code. Another option is Sip, a relative of SWIG, which focuses on C++. The Boost.Python library is yet another tool that can be used to write small bits of code to create a shared library. Of these three, SWIG is the most popular, probably because it plays well not only with C, C++, Python, and Ruby, but also with Perl, Tcl/Tk, Java, and C#. SWIG is copyrighted software, but it is freely distributed. It is normally found on UNIX but will also operate on Win32 OSs.

SWIG automates the wrapper process by generating wrapper code from a list of ANSI C functions and variable declarations. The SWIG language is actually fairly complex and very complete. It supports preprocessing, pointers, classes, inheritance, and even C++ templates.

SWIG is typically called from a command prompt or used with NMAKE. Modules can be compiled into a DLL form and then dynamically loaded into Python, or they can be set up as a custom build option in MS Development Studio. SWIG can be found online at Sourceforge (http://swig.sourceforge.net/), and Boost.Python, by David Abrahams, can be found online at Python.org (http://www.python.org/cgi-bin/moinmoin/boost_2epython).

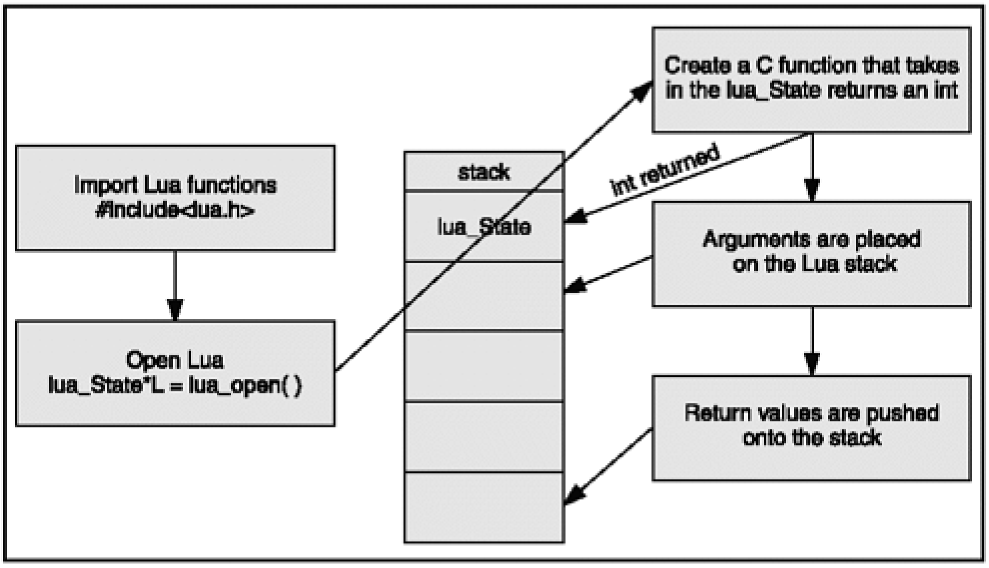
Extending Lua

Lua was built to partner with other languages, and it can be extended with functions written in C just as Python can. These functions must be of the `lua_CFunction` type:

```
typedef int (*lua_CFunction) (lua_State *L);
```

A C function receives a Lua state and returns an integer that holds the number of values that must return to Lua (see Figure 12.5). The C function receives arguments from Lua in its stack in direct order. Any return values to Lua are pushed onto the stack, also in direct order.

Figure 12.5. Representation of Lua and C partnership



When registering a C function to Lua, a built-in macro receives the name the function will have in Lua and a pointer to the function, so a function can be registered in Lua by calling the `lua_register` macro:

```
lua_register(L, "average", MyFunction);
```

Values can be associated with a C function when it is created. This creates what is called a C closure. The values are then accessible to the C function whenever it is called. To create a C closure, first push the values onto the stack, and then use the

`lua_pushcclosure` command to push the C function onto the stack with an argument containing the number of values that need to be associated with the function:

```
void lua_pushcclosure (lua_State *L, lua_CFunction MyFunction, int
MyArgument);
```

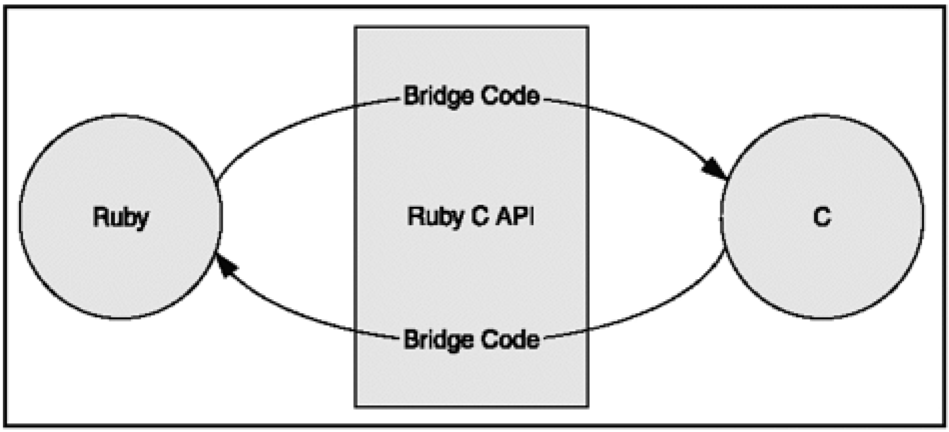
Whenever the C function is called, the values pushed up are located at specific pseudo-indices produced by a macro, `lua_upvalueindex`. The first value is at position `lua_upvalueindex(1)`, the second at `lua_upvalueindex(2)`, and so on.

Lua also provides a predefined table that can be used by any C code to store whatever Lua value it needs to store. This table is a registry and is really useful when values must be kept outside the lifespan of a given function. This registry table is pseudo-indexed at `LUA_REGISTRYINDEX`. Any C library can store data into this table.

Extending Ruby

Extending Ruby in C is accomplished by writing C as a bridge between Ruby's C API and whatever you want to add on to Ruby (see Figure 12.6). The Ruby C API is contained in the C header file ruby.h, and many of the common API commands are listed in Table 12.2.

Figure 12.6. The Ruby C API



Ruby and C must share data types, which is problematic when Ruby only recognizes objects. For C to understand Ruby, some translation must be done with data types. In Ruby, everything is either an object or a reference to an object. For C to understand Ruby, data types must be pointers to a Ruby object or actual objects. You do so by making all Ruby variables in C a `VALUE`type. When `VALUE` is a pointer, it points to one of the memory structures for a Ruby class or object structure. `VALUE` can also be an immediate value such as `Fixnum`, `Symbol`, `true`, `false`, or `nil`.

A Ruby object is an allocated structure in memory that contains a table of instance variables and other class information. The class is another allocated structure in memory

that contains a table of the methods defined for that class. The built-in objects and classes are defined in the C API's header file, ruby.h. Before wrapping up any Ruby in C, you must include this file:

```
#include "ruby.h"
```

You must define a C global function that begins with `Init_` when writing new classes or modules. Creating a new subclass of Ruby's object looks like the following:

```
void Init_MyNewSubclass() {
        cMyNewSubclass = rb_define_class("MyNewSubclass", rb_cObject);
        }
```

`Object`is represented by `rb_cObject` in the ruby.h header file, and the class is defined with `rb_define_class`. Methods can be added to the class using `rb_define_method`, like so:

```
void Init_MyNewSubclass() {
        cMyNewSubclass = rb_define_class("MyNewSubclass", rb_cObject);
        rb_define_method(cMyNewSubclass, "MyMethod", MyFunction, value
);
        }
```

Ruby and C can also directly share global values. This is accomplished by first creating a Ruby object in C:

```
VALUE MyString;
MyString = rb_str_new();
```

Then bind the object's address to a Ruby global variable:

```
Rb_define_variable("$String", &MyString);
```

Now Ruby can access the C variable `MyString` as `$String`.

You may run into trouble with Ruby's garbage collection when extending Ruby. Ruby's GC needs to be handled with kid gloves when C data structures hold Ruby objects or when Ruby objects hold C structures. You can smooth the way by writing a function that registers the objects, passing `free()`, calling `rb_global_variable()` on each Ruby object in a structure, or making other special API calls.

Once code has been written for an extension, it needs to be compiled in a way that Ruby can use. The code can be compiled as a shared object to be used at runtime, or it can be statically linked to the Ruby interpreter. The entire Ruby interpreter can also be embedded within an application. The steps you should take depend greatly on the platform on which the programming is being done; there are instructions for each

method on the online Ruby library reference, at http://www.ruby-lang.org/en/20020107.html.

The C API, however, is quite large, and for English users the best source for documentation is likely the source code itself.

Table 12.2. Common Ruby C Language APIs

Type	API Command	Function
`char`	`rb_id2name()`	Returns a name for the given ID
`ID`	`rb_intern()`	Returns an ID for a given name
`int`	`Check_SafeStr()`	For raising `SecurityError`
`int`	`OBJ_FREEZE()`	Marks the given object as frozen
`int`	`OBJ_FROZEN()`	For testing if an object is frozen
`int`	`OBJ_TAINT()`	Marks the given object as tainted
`int`	`OBJ_TAINTED()`	For testing if an object is tainted
`int`	`rb_block_given_p()`	Returns `true` if yield would execute a block in the current context
`int`	`rb_cvar_defined()`	Returns `Qtrue` if the given class variable name has been defined, otherwise returns `Qfalse`
`int`	`rb_safe_level()`	Returns the current safe level
`int`	`rb_scan_args()`	Scans the argument list and assigns them in a similar way to `scanf`
`int`	`rb_secure()`	Raises `SecurityError` if level is less than or equal to the current safe level
`VALUE`	`rb_apply()`	Function for invoking methods
`VALUE`	`rb_ary_entry()`	Returns an array element at a given index
`VALUE`	`rb_ary_new()`	Returns a new array
`VALUE`	`rb_ary_new2()`	Returns a new (long) array
`VALUE`	`rb_ary_new3()`	Returns a new array populated with the given arguments
`VALUE`	`rb_ary_new4()`	Returns a new array populated with the given C array values
`VALUE`	`rb_ary_push()`	Pushes a value onto the end of an array `self`
`VALUE`	`rb_ary_pop()`	Removes and returns the last element from an array
`VALUE`	`rb_ary_shift()`	Removes and returns the first element from an array

Table 12.2. Common Ruby C Language APIs

Type	API Command	Function
`VALUE`	`rb_ary_unshift()`	Pushes a value onto the front of an array `self`
`VALUE`	`rb_call_super()`	Calls the current method in the super class of the current object
`VALUE`	`rb_catch()`	Equivalent to Ruby `catch`
`VALUE`	`rb_cv_get()`	Returns class variable name
`VALUE`	`rb_cvar_get()`	Returns the class variable name from the given class
`VALUE`	`rb_define_class()`	Defines a new top-level class
`VALUE`	`rb_define_class_under()`	Defines a nested class
`VALUE`	`rb_define_module()`	Defines a new top-level module
`VALUE`	`rb_define_module_under()`	Defines a nested module
`VALUE`	`rb_each()`	Invokes the `each` method of the given object
`VALUE`	`rb_funcall()`	Invokes methods
`VALUE`	`rb_funcall2()`	Invokes methods
`VALUE`	`rb_funcall3()`	Invokes methods
`VALUE`	`rb_gv_get()`	Returns the global variable name
`VALUE`	`rb_gv_set()`	Sets the global variable name
`VALUE`	`rb_hash_aref()`	Returns element corresponding to given key
`VALUE`	`rb_hash_aset()`	Sets the value for a given key
`VALUE`	`rb_hash_new()`	Returns a new hash
`VALUE`	`rb_iterate()`	Invokes method with given arguments and block
`VALUE`	`rb_ivar_get()`	Returns the instance variable name from the given object
`VALUE`	`rb_ivar_set()`	Sets the value of the instance variable name in the given object to a given value
`VALUE`	`rb_iv_get()`	Returns the instance variable name
`VALUE`	`rb_iv_set()`	Sets the value of the instance variable name
`VALUE`	`rb_rescue()`	Executes until a `StandardError` exception is raised, then executes `rescue`
`VALUE`	`rb_str_dup()`	Returns a new duplicated string object

Table 12.2. Common Ruby C Language APIs

Type	API Command	Function
VALUE	rb_str_cat()	Concatenates length characters on string
VALUE	rb_str_concat()	Concatenates other on string
VALUE	rb_str_new()	Returns a new string initialized with length characters
VALUE	rb_str_new2()	Returns a new string initialized with null-terminated C string
VALUE	rb_str_split()	Splits a string at the given deliminator and returns an array of the string objects
VALUE	rb_thread_create()	Runs a given function in a new thread
VALUE	rb_yield()	Transfers execution to the iterator block in the current context
void	rb_ary_store()	Stores a value at a given index in an array
void	rb_bug()	Terminates the process immediately
void	rb_cvar_set()	Sets the class variable name in the given class to `value`
void	rb_cv_set()	Sets the class variable name
void	rb_define_alias()	Defines an alias in a class or module
void	rb_define_attr()	Creates access methods for the given variable with the given name
void	rb_define_class_variable()	Defines a class variable name
void	rb_define_const()	Defines a constant in a class or module
void	rb_define_global_const()	Defines a global constant
void	rb_define_global_function()	Defines a global function
void	rb_define_hooked_variable()	Defines functions to be called when reading or writing to variable
void	rb_define_method()	Defines an instance method
void	rb_define_module_function()	Defines a method in the given class module with the given name
void	rb_define_readonly_variable()	Same as `rb_define_variable` except is read-only from Ruby
void	rb_define_singleton_method()	Defines a singleton method
void	rb_define_variable()	Exports the address of the given object that was created in C to the Ruby namespace as a given name
void	rb_define_virtual_variable()	Exports a virtual variable to the Ruby namespace

Table 12.2. Common Ruby C Language APIs

Type	API Command	Function
`void`	`rb_exit()`	Exits Ruby with the given status
`void`	`rb_extend_object()`	Extends given object with module
`void`	`rb_fatal()`	Raises a fatal exception
`void`	`rb_include_module()`	Includes the given module into the class or module parent
`void`	`rb_iter_break()`	Breaks out of the enclosing iterator block
`void`	`rb_notimplement()`	Raises a NotImpError exception
`void`	`rb_raise()`	Raises an exception
`void`	`rb_set_safe_level()`	Sets the current safe level
`void`	`rb_sys_fail()`	Raises a platform-specific exception
`void`	`rb_throw()`	Equivalent to Ruby `throw`
`void`	`rb_undef_method()`	Undefines the given method name in the given class or module
`void`	`rb_warn()`	Unconditionally issues a warning message to standard error
`void`	`rb_warning()`	Conditionally issues a warning message to standard error

Python versus Lua Versus Ruby

So which of the three languages is the best to use on your project? That depends a great deal on what you want to accomplish. To wrap up the book, I've outlined some of the pros and cons of each language in this section.

Python Pros and Cons

The pros of Python are as follows:

- Python has more extension modules than the other languages.
- Many online Python tutorials exist. There are also plenty of English books and reference materials, many sample scripts exist online, and there is a wealth of introductory material. The Python.org Website is a good place to start looking for these because it has sections for beginners, tutorials, guides organized by topic, and lists of links and references.
- Most folks really enjoy the syntax of the Python language because it appears clean and is easy to read.
- Python has an edge where libraries are concerned. There are many libraries, and, for the most part, they are well documented.
- Lots of tools that tie into Python are available, and they are often easier to find than the tools for Lua and Ruby.

The cons of Python are as follows:

- Existing Python debuggers are considered quirky and slow. Debugging support on Macintosh and consoles is even weaker.
- It can be difficult to bundle Python with other languages. There are lots of binary DLLs, and Python has (compared to the other languages) a large standard distribution.
- Lots of folks really dislike the white space sensitivity of Python syntax.
- Python can be quite slow at times, as everything is an object on the heap.

Lua Pros and Cons

The pros of Lua are as follows:

- Lua is probably the fastest of the three languages and usually uses the least amount of runtime memory.
- Lau has the smallest memory footprint for bundling.
- The Lau C API is very well documented and has good examples for integrating with C.

The cons of Lua are as follows:

- The documentation has improved but is still a bit sketchy overall. Of the three languages, Lua it is probably the least documented (the API being the exception), with the least amount of code comments. This makes for the largest ramp-up time to learn, and there isn't much in the way of introductory Lua material.

- There isn't a lot of built-in functionality for Lua. There is little support if you need to create a large, complex application.
- Lua could use a better garbage collector—the current development is moving towards that now. Right now, Lua GC uses a very simple and traditional simple mark and sweep.

Pros and Cons of Ruby

The pros of Ruby are as follows:

- Ruby possesses fairly good advanced debuggers.
- Ruby is object oriented from the ground up, and programmers who are OOP enthusiasts or who are used to the OOP paradigm will find the language extremely comfortable.
- Ruby has arguably the simplest syntax, with no real rules exceptions. Especially true for OOP enthusiasts.

The cons of Ruby are as follows:

- Lack of English documentation.
- Fewer existing works and samples for games than with the other languages.

Summary

Programming is turning more and more into an everyman's tool. Every single day, software becomes easier for everyone to use. High-level languages are behind this incredible movement in Game programming. Today, because of these incredible languages, games are released with hooks, customizable engines, their own languages, and modifiable graphics. This can be accomplished in development using a data-driven, partnered game design model.

A few important points to take from this chapter:

- There are a number of things to consider before including high-level languages in a development project.
- Extending a high-level language can allow two or more languages to really focus on what they are good at in a single project, but it adds a layer of complexity.
- Extending is a similar process in Python, Lua, and Ruby.

Exercises

1: How do you call an object's method from C using Python? Lua? Ruby?

2: Write sample code to extract C values from one of the three languages' object. Watch out for types!

3: List two possible issues when using extensions in a project.

Appendix A. History of Computer Programming

The following table outlines the history of computer programming through its (arguably, in some cases) most important events.

Table A.1. A Brief History of Programming

Year	Event
Around 4000 BC	Clay tablets are used to keep track of transactions.
Around 3000 BC	Abacus invented in Babylonia.
Around 800 AD	The Chinese start to use the number 0, although some historians believed it was introduced from India.
1612-1617	John Napier uses the decimal point, devises logarithms, and uses numbered sticks for calculation.
1622	William Oughtred invents the circular slide rule based on Napier's logarithms.
1786	J.H.Mueller dreams up his "Difference Engine," but like many dot com companies, he cannot get the funds from investors to build it.
1822	Charles Babbage begins to redesign and build Mueller's Difference Engine with funding from the British government.
1834-35	Babbage changes his focus from the Difference Engine to a new version called the Analytical Engine.
1840s	Ada Lovelace becomes the world's first programmer by putting together methods of computing using Babbage's notes on the Analytical Engine.
1842	The British government pulls funding for the construction of the Difference Engine.
1847-49	Babbage completes 21 drawings for a new improved second version of the Difference Engine but still does not complete construction.
1853	The Difference Engine is finally completely built, but by another group not including Babbage.
1854	Herman Hollerith, whose electric tabulating system was used for the 1890 census, establishes the Tabulating Machine Company. TMC will later become IBM.
1941	Atanasoff and Berry build the first electronic (and non-programmable) computer named ABC. Zuse completes the Z3 machine, the world's first fully functional program in an automatically controlled electro-mechanical computer. It has a 64-word memory and computes at three seconds per multiplication.
1944	Howard Aiken completes the first programmable computer, the Mark I, using punched paper tape for programming and vacuum tubes and relays to

Table A.1. A Brief History of Programming

Year	Event
	calculate problems.
1945	Zuse develops "Plankalkul" (short for plain calculus), which is considered the first programming language and was designed to be a chess-playing (i.e. game) program. Also, on Sept 9th, working on a prototype of the Mark II, Grace Murray finds the first computer "bug," an actual moth that caused a relay failure.
1951	Betty Holberton creates a "Sort Merge Generator," a predecessor to modern compilers.
1957	FORTRAN appears, short for Mathematical FORmula TRANslating System. Heading the FORTRAN team is John Backus, who also goes on to contribute to the development of ALGOL and BNF.
1958	John McCarthy introduces the Lisp programming language.
1958	First computers to be built with transistors instead of vacuum tubes.
1959	There are now over 200 programming languages in existence.
1960	COBOL, created by the Conference on Data Systems and Languages, is launched for business applications.
1962	Spacewar, arguably the first video game ever, is invented at MIT by a graduate student named Steve Russel.
1964	At Dartmouth University, professors John G.Kemeny and Thomas E. Kurtz invent BASIC. The first BASIC program runs on May 1, 1964 (at around 4 a.m.).
1965	Ken Iverson develops the APL language at IBM.
1967	IBM announces that it will no longer bundle software and hardware together, but rather will sell them separately. This business move is considered the beginning of the software industry.
1968	Edsgar Dijkstra first writes about the harmful effects of the `goto` statement. Intel is formed and incorporated on July 18th.
1968	ALTRAN, a FORTRAN variant, appears. COBOL is officially defined by ANSI.
1969	Kenneth Thornson and Dennis Ritchie formulate UNIX at AT&T Bell Labs. Donald Knuth writes Volume 1 of the Art of Computer Programming, considered the first computer programming book.
1971	Niklaus Wirth develops Pascal, a predecessor of Modula-2.
1972	Nolan Buchnell's game Pong is so popular that he founds Atari.Rary Tarnlinson creates e-mail to send personal messages across Arpnet (Arpnet will become the Internet;currently it is used only by the military). Smalltalk is developed by Xerox PARC's learning research group. Denis Ritchie develops C at Bell Labs.

Table A.1. A Brief History of Programming

Year	Event
1975	The Altair 8800 is available in January as a kit you can order and build from Popular Mechanics, and the PC is born.Bill Gates and Paul Allen write a version of BASIC that they sell to MITS (Micro Instrumentation and Telemetry Systems) on a per-copy royalty basis.Scheme, a Lisp dialect by G.L.Steele and G.J.Sussman, appears.
1976	Crowther and Woods create the first adventure game called—you guessed it—Adventure. Steve Jobs and Steve Wozniak design and build the Apple I.
1977	Bill Gates and Paul Allen found Microsoft in Albuquerque, New Mexico.
1979	Pac Man appears.
1980	IBM selects PC-DOS from the Microsoft Corporation as the operating system for its new PC. Smalltalk-80 appears.Bjarne Stroustrup develops a set of languages, collectively referred to as "C With Classes," which serves as the breeding ground for C++.
1981	Japan begins the Fifth Generation Computer System project using Prolog as the primary language.
1983	Microsoft announces "Windows," a graphical user interface for PCs. Windows doesn't actually ship, however, until 1985. The first C compilers for microcomputers are released. In July the first implementation of C++ appears.
1984	The Macintosh is unveiled, with much glitter and hype, at the Super Bowl.William Gibson coins the term "cyberspace" in his novel Neuromancer.
1985	Windows finally launches. The C++ language is issued from Bell Labs. The Intel 80386 chip with 32-bit processing is released.
1986	The programming language Eiffel appears.
1987	The Perl programming language is released.
1989	The C programming language is standardized by ANSI.
1990	By now more than 54 million computers are in use in the United States alone, and the first commercially available dial-up Internet access appears.
1991	The Python programming language is released.
1992	The programming language Dylan is released by Apple.
1993	The Ruby programming language is released.
1994	The Lua programming language is released.Netscape's first browser becomes available.
1995	Sun Microsystems releases Java.
1996	One out of every three homes in the United States has a computer.

Appendix B. Meet the Family

After the first high-level languages were developed in the 1950s, dozens of other languages popped up and followed suit. Today, you can't surf the Web or sit on a busy subway without encountering them in use in some form or another. This book focuses on three languages most commonly used in game shops, but there are dozens of others in popular use.

ABC

Created by Leo Geurts, Lambert Meertens, and Steven Pemberton. The idea behind ABC was to create a simple, interactive language designed for quick and easy programming. ABC was originally intended to replace BASIC.

Ada

Ada was developed in the 1970s by the United States Department of Defense. Named after Lady Ada Lovelace Byron, Ada is a general-purpose language used for everything from business apps to rocket science. Ada is mandatory for the development of many major U.S. military projects and has been used for large real-time systems for air-traffic control and banking.

AFNOR

AFNOR isn't actually a language, but a standards-setting organization. AFNOR is an acronym for Association Français Normal and is part of the International Organization for Standardization that also includes ANSI (American National Standards Institute), the BSI (British Standards Institution), DIN (Deutsche Institut für Normung), and other standards organizations.

C

C is credited to Dennis Ritchie at Bell Labs in 1972. C was originally a systems language for UNIX on the PDP-11 and was briefly named NB. Partly due to its free distribution with UNIX, C became the language most widely used for software implementation. C has gone through a few incarnations, including K&R (Kernighan and Ritchie) C, and ANSI C, and has been lately revamped as the object-oriented C++.

C++

Both C and C++ are considered high-level languages, although they are much closer to machine assembly than other high-level languages. This makes them very efficient but sometimes difficult to implement. C++ was developed at Bell Labs by Bjarne Strousrup, who took C and added object-oriented programming (OOP) features. The C family is especially brilliant when it comes to creating the very popular graphics and Windows-based applications and has a wonderful section of well-designed libraries.

Cobol

Cobol is short for Common Business Oriented Languages. Cobol goes way back to the 1950s and is considered one of the old timers (with FORTRAN being its father). Cobol's focus was, of course, business applications that ran on large computers. Back in the 1950s Cobol wasn't really considered high-level, it was considered wordy. The wordiness makes it easy to follow the business jargon, but it also requires a lot more typing than other languages.

Eiffel

Released by Bertrand Meyer in 1986. Eiffel is considered an object-oriented language, has automatic garbage collection, and possesses interfaces to routines written in other languages. It is implemented as a C preprocessor.

FORTRAN

FOTRAN is an acronym for FORmula TRANslator. It is probably the oldest high-level language, originally designed at IBM by John Backus in the late 1950s. The language has branched into several different versions, many of which are still in use today. FORTRAN's niche is mathematical computations, and it is most commonly used in universities.

GNU Octave

Used for numerical computations, GNU Octave has lots of tools for common math and algebra functions and tasks. GNU Octave is customizable, can run via command line or through batch, and can dynamically load up FORTRAN or C for other tasks. GNU Octave is distributed under the GNU General Public License published by the Free Software Foundation.

Java

Originally developed by Sun Microsystems for set-top boxes and handheld devices in an incarnation known as Oak, Java moved to the World Wide Web in 1995 and took off because it was multi-platform. Java is similar to C++ but was designed with OOP and security in mind from the ground up, and efforts were made in its structure to remove features that caused common errors and bugs (like pointers and garbage collection).

Icon

Icon is another high-level language used often in research and text processing. Icon was developed at the University of Arizona and is loosely based on Bell Lab's Snobol.

Modula

Short for MODUlar LAnguage, Modula precedes Modula-2, developed as a system language for the Lilith workstation. The central concept behind Modula is the module—a programming construct that can be used to encapsulate a set of related subprograms and data structures. Modules are also restricted in their visibility from other portions of the program. Modula-2 precedes Modula-2+ and Modula-3.

Pascal

Pascal was developed in the late 1960s by Niklaus Wirth and was named after Blaise Pascal, who was a 17th-century French mathematician who constructed early adding machines. In addition to being high-level, Pascal is also a structured programming language, which forces design into its very nature. Pascal is often used as a teaching tool because of its regimented structure.

Perl

Short for Practical Extraction and Report Language, Perl was released in 1987 by Larry Wall, who developed the language while working for the National Security Agency. Larry wanted his language to be based on common sense programming techniques and wanted applications developed with Perl to be quickly and easily written. Perl was built originally as a simple language to scan text files, extract information from those files, and print reports based on that information. It has blossomed into a full programming language with hundreds of supplemental libraries. Perl is easy to learn and is commonly found on the Internet, used in conjunction with CGI and HTML.

PHP

PHP is a domain-specific language for Web server-side scripting. PHP embeds itself into HTML to create dynamic Web pages. The language has a syntax similar to Perl's or C's and is comparable to CGI; its primary strength is in database access. PHP was originally developed in 1994 but has gone through at least one major rewrite and has had many contributors.

Prolog

Short for PROgramming LOGic, Prolog is a high-level language based on the discipline of traditional logic. While most computer languages perform a sequence of commands, Prolog has an entirely different approach. Prolog first creates definitions and assumptions and then uses them to solve logic problems. For Prolog, a program is just a list of facts and rules. Prolog is most often found in AI experiments and expert systems (programs that function like human experts).

PureBasic

A high-level language based on BASIC, a revival of sorts that focuses on keeping programming linear and simple. PureBasic is a good learning tool with a few games under its belt, including Bricklinerby Wegroup, Krakout 2 Unlimited(a remake of the Commodore 64 game Krakout), and a few titles by Reelmedia.

Smalltalk

Smalltalk was created by Software Concepts Group (i.e.Xerox) in a development led by Alan Kay in the early 1970s. Smalltalk took the concepts of class and message from Simula-67 and made them pervasive, basically creating the quintessential object-oriented language. Early versions were Smalltalk-72, Smalltalk-74, and Smalltalk-76; now we're on Smalltalk-80.

Squeak

Disney and Paul Allen's Interval Research Lab helped develop the open source Squeak language. Squeak has three environments: one for young children, one for middle school through adult age, and one for experts who are into "deep computing."